Robotic Manipulation

Perception, Planning, and Control

Russ Tedrake

Last modified 2023-1-15.
How to cite these notes, use annotations, and give feedback.

Note: These are working notes used for a course being taught at MIT. They will be updated throughout the Fall 2022 semester.

PDF version of the notes

The PDF version of these notes are autogenerated from the HTML version. There are a few conversion/formatting artifacts that are easy to fix (please feel free to point them out). But there are also interactive elements in the HTML version are not easy to put into the PDF. When possible, I try to provide a link. But I consider the online HTML version to be the main version.

Table of Contents

PREFACE

I've always loved robots, but it's only relatively recently that I've turned my attention to robotic manipulation. I particularly like the challenge of building robots that can master physics to achieve human/animal-like dexterity and agility. It was passive dynamic walkers and the beautiful analysis that accompanies them that first helped cement this centrality of dynamics in my view of the world and my approach to robotics. From there I became fascinated with (experimental) fluid dynamics, and the idea that birds with articulated wings actually "manipulate" the air to achieve incredible efficiency and agility. Humanoid robots and fast-flying aerial vehicles in clutter forced me to start thinking more deeply about the role of perception in dynamics and control. Now I believe that this interplay between perception and dynamics is truly fundamental, and I am passionate about the observation that relatively "simple" problems in manipulation (how do I button up my dress shirt?) expose the problem beautifully.

My approach to programming robots has always been very computational/ algorithmic. I started out using tools primarily from machine learning (especially reinforcement learning) to develop the control systems for simple walking machines; but as the robots and tasks got more complex I turned to more sophisticated tools from model-based planning and optimization-based control. In my view, no other discipline has thought so deeply about dynamics as has control theory, and the algorithmic efficiency and guaranteed performance/robustness that can be obtained by the best model-based control algorithms far surpasses what we can do today with learning control. Unfortunately, the mathematical maturity of controls-related research has also led the field to be relatively conservative in their assumptions and problem formulations; the requirements for robotic manipulation break these assumptions. For example, robust control typically assumes dynamics that are (nearly) smooth and uncertainty that can be represented by simple distributions or simple sets; but in robotic manipulation, we must deal with the non-smooth mechanics of contact and uncertainty that comes from varied lighting conditions, and different numbers of objects with unknown geometry and dynamics. In practice, no state-of-the-art robotic manipulation system to date (that I know of) uses rigorous control theory to design even the low-level feedback that determines when a robot makes and breaks contact with the objects it is manipulating. An explicit goal of these notes is to try to change that.

In the past few years, deep learning has had an unquestionable impact on robotic perception, unblocking some of the most daunting challenges in performing manipulation outside of a laboratory or factory environment. We will discuss relevant tools from deep learning for object recognition, segmentation, pose/keypoint estimation, shape completion, etc. Now relatively old approaches to learning control are also enjoying an incredible surge in popularity, fueled in part by massive computing power and increasingly available robot hardware and simulators. Unlike learning for perception, learning control algorithms are still far from a technology, with some of the most impressive looking results still being hard to understand and to reproduce. But the recent work in this area has unquestionably highlighted the pitfalls of the conservatism taken by the controls community. Learning researchers are boldly formulating much more aggressive and exciting problems for robotic manipulation than we have seen before -- in many cases we are realizing that some manipulation tasks are actually quite easy, but in other cases we are finding problems that are still fundamentally hard.

Finally, it feels that the time is ripe for robotic manipulation to have a real and dramatic impact in the world, in fields from logistics to home robots. Over the last few years, we've seen UAVs/drones transition from academic curiosities into

consumer products. Even more recently, autonomous driving has transitioned from academic research to industry, at least in terms of dollars invested. Manipulation feels like the next big thing that will make the leap from robotic research to practice. It's still a bit risky for a venture capitalist to invest in, but nobody doubts the size of the market once we have the technology. How lucky are we to potentially be able to play a role in that transition?

So this is where the notes begin... we are at an incredible crossroads between learning and control and robotics with an opportunity to have immediate impact in industrial and consumer applications and potentially even to forge entirely new eras for systems theory and controls. I'm just trying to hold on and to enjoy the ride.

A MANIPULATION TOOLBOX

Another explicit goal of these lecture notes is to provide high-quality implementations of the most useful tools in a manipulation scientist's toolbox. When I am forced to choose between mathematical clarity and runtime performance, the clear formulation is always my first priority; I will try to include a performant formulation, too, if possible or try to give pointers to alternatives. Manipulation research is moving quickly, and I aim to evolve these notes to keep pace. I hope that the software components provided in DRAKE and in these notes can be directly useful to you in your own work.

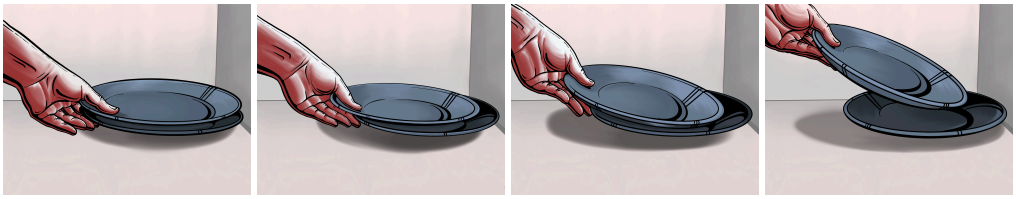
If you would like to replicate any or all of the hardware that we use for these notes, you can find information and instructions in the appendix.

As you use the code, please consider contributing back (especially to the mature code in DRAKE). Even questions/bug reports can be important contributions. If you have questions/find issues with these notes, please submit them here.

First chapter

Chapter 1
Introduction

It's worth taking time to appreciate just how amazingly well we are able to perform tasks with our hands. Tasks that often feel mundane to us -- loading the dishwasher, chopping vegetables, folding laundry -- remain as incredibly challenging tasks for robots and are at the very forefront of robotics research.



Consider the problem of picking up a single plate from a stack of plates in the sink and placing it into the dishwasher. Clearly you first have to perceive that there is a plate in the sink and that it is accessible. Getting your hand to the plate likely requires navigating your hand around the geometry of the sink and other dishes. The act of actually picking it up might require a fairly subtle maneuver where you have to tip up the plate, sliding it along your fingers and then along the sink/dishes in order to get a reasonable grasp on it. Presumably as you lift it out of the sink, you'd like to mostly avoid collisions between the plate and the sink, which suggests a reasonable understanding of the size/extent of the plate (though I actually think robots today are too afraid of collisions). Even placing the plate into the dishwasher is pretty subtle. You might think that you would align the plate with the slats and then slide it in, but I think humans are more clever than that. A seemingly better strategy is to loosen your grip on the plate, come in at an angle and intentionally contact one side of the slat, letting the plate effectively rotate itself into position as you set it down. But the justification for this strategy is subtle -- it is a way to achieve the kinematically accurate task without requiring much kinematic accuracy on the position/orientation of the plate.

Figure 1.2 - A robot picking up a plate from a potentially cluttered sink (left: in simulation, right: in reality). This example is from the manipulation team at the Toyota Research Institute.

Perhaps one of the reasons that these problems remain so hard is that they require strong capabilities in numerous technical areas that have traditionally been somewhat disparate; it's challenging to be an expert in all of them. More so than robotic mapping and navigation, or legged locomotion, or other great areas in robotics, the most interesting problems in manipulation require significant interactions between perception, planning, and control. This includes both geometric perception to understand the local geometry of the objects and environment and semantic perception to understand what opportunities for manipulation are available in the scene. Planning typically includes reasoning about the kinematic and dynamic constraints of the task (how do I command my rigid seven degree-of-freedom arm to reach into the drawer?). But it also includes higher-

level task planning (to get milk into my cup, I need to open the fridge, then grab the bottle, then unscrew the lid, then pour... and perhaps even put it all back when I'm done). The low-level begs for representations using real numbers, but the higher levels feel more logical and discrete. At perhaps the lowest level, our hands are making and breaking contact with the world either directly or through tools, exerting forces, rapidly and easily transitioning between sticking and sliding frictional regimes -- these alone are incredibly rich and difficult problems from the perspective of dynamics and control.

There is a lot for us to discuss!

1.1 Manipulation is more than pick-and-place

There are a large number of applications for manipulation. Picking up an object from one bin and placing it into another bin -- one version of the famous "pick and place" problem -- is a great application for robotic manipulation. Robots have done this for decades in factories with carefully curated parts. In the last few years, we've seen a new generation of pick-and-place systems that use deep learning for perception, and can work well with much more diverse sets of objects, especially if the location/orientation of the placement need not be very accurate. This can be done with conventional robot hands or more special-purpose end-effectors that, for instance, use suction. It can often be done without having a very accurate understanding of the shape, pose, mass, nor friction of the object(s) to be picked.

The goal for these notes, however, is to examine the much broader view of manipulation than what is captured by the pick and place problem. Even our thought experiment of loading the dishwasher -- arguably a more advanced type of pick and place -- requires much more from the perception, planning, and control systems. But the diversity of tasks that humans (and hopefully soon robots) can do with their hands is truly remarkable. To see one small catalog of examples that I like, take a look at the Southampton Hand Assessment Procedure (SHAP), which was designed as a way to empirically evaluate prosthetic hands. Matt Mason also gave a broad and thoughtful definition of manipulation in the opening of his 2018 review paper[1].

It's also important to recognize that manipulation research today looks very different than manipulation research looked like in the 1980s and 1990s. During that time there was a strong and beautiful focus on "manipulation as grasping," with seminal work on, for instance, the kinematics/dynamics of multi-fingered hands assuming a stable grasp on an object. Sometimes still, I hear people use the term "grasping" as almost synonymous with manipulation. But please realize that the goals of manipulation research today, and of these notes, are much broader than that. Is grasping a sufficient description of what your hand is doing when you are buttoning your shirt? Making a salad? Spreading peanut butter on toast?

1.2 Open-world manipulation

Perhaps because humans are so good at manipulation, our expectations in terms of performance and robustness for these tasks are extremely high. It's not enough to be able to load one set of plates in a laboratory environment into a dishwasher reliably. We'd like to be able to manipulate basically any plate that someone might put in the sink. And we'd like the system to be able to work in any kitchen, despite various geometric configurations, lighting conditions, etc. The challenge of achieving and verifying robustness of a complex manipulation stack with physics, perception, planning, and control in the loop is already daunting. But how do we provide test coverage for every kitchen in the world?

The idea that the world has infinite variability (there will never be a point at which you have seen every possible kitchen) is often referred to as the "open-world" or "open-domain" problem -- a term popularized first in the context of video games. It can be tough to strike a balance between rigorous thinking about aspects of the manipulation problem, and trying to embrace the diversity and complexity of the

entire world. But it's important to walk that line.

There is a chance that the diversity of manipulation in the open world might actually make the problem easier. We are just at the dawn of the era of big data in robotics; the impact this will have cannot be overstated. But I think it is deeper than that. As an example, some of our optimization formulations for planning and control might get stuck in local minima now because narrow problem formulations can have many quirky solutions; but once we ask a controller to work in a huge diversity of scenarios then the quirky solutions can be discarded and the optimization landscape may become much simpler. But to the extent that is true, then we should aim to understand it rigorously!

1.3 Simulation

There is another reason that we are now entering a golden age for manipulation research. Our software tools have (very recently) gotten much better!

I remember a just few years ago (~2015) talking to my PhD students, who were all quite adept at using simulation for developing control systems for walking robots, about using simulation for manipulation. "You can't work on manipulation in simulation" was their refrain, and for good reason. The complexity of the contact mechanics in manipulation has traditionally been much harder to simulate than a walking robot that only interacts with the ground and through a minimal set of contact points. Looming even larger, though, was the centrality of perception for manipulation; it was generally accepted that one could not simulate a camera well enough to be meaningful.

How quickly things can change! The last few years has seen a rapid adoption of video-game quality rendering by the robotics and computer vision communities. The growing consensus now is that game-engine renderers *can* model cameras well enough not only to *test* a perception system in simulation, but even to *train* perception systems in simulation and expect them to work in the real world! This is fairly amazing, as we were all very concerned before that training a deep learning perception system in simulation would allow it to exploit any quirks of the simulated images that could make the problem easier.

We have also seen dramatic improvements in the quality and performance of contact simulation. Making robust and performant simulations of multi-body contact involves dealing with complex geometry queries and stiff (measure-) differential equations. There is still room for fundamental improvements in the mathematical formulations of and numerical solutions for the governing equations, but today's solvers are good enough to be extremely useful.

1.4 These notes are interactive

By leveraging the power of simulation, and the new availability of free online interactive compute, I am trying to make these notes more than what you would find in a traditional text. Each chapter will have working code examples, which you can run immediately (no installation required) on Deepnote, or download/install and run on your local machine (see the appendix for more details). It uses an open-source library called Drake which has been a labor of love for me since around 2013 when I started taking our research code and making it more broadly available. Because all of the code is open source, it is entirely up to you how deep into the rabbit hole you choose to go.

Mortimer Adler famously said "Reading a [great] book should be a conversation between you and the author." In addition to the interactive graphics/code, I've added the ability to highlight/comment/ask questions directly on these notes (go ahead and try it; but please read my note on annotation etiquette). Adler's suggestion was that *great* writing can turn static text into a dialogue, transcending distance and time; perhaps I'm cheating, but technology can help me communicate with you even if my

writing isn't as strong as Adler would have liked! Adler also recommends writing on your books[2]; you can print the pages, use the (infrequently updated) pdf, or make private annotations right on the website using the same annotation tool.

I have organized the software examples into notebooks by chapter. There is an "Launch in Deepnote" button at the top of each chapter; I'd encourage you to open it immediately when you are reading the chapter. Go ahead and "duplicate" the chapter project and run the first cell to startup the cloud machine. Then as you read the text, I will have examples that will have corresponding sections in the notebook.

Example 1.1 (Teleoperation in 2D.)

Before we get into any autonomous manipulation, let's start by just getting a feel for what it will be like to work on manipulation in an online Jupyter notebook. This example will open up a new window with our 3D visualizer, and in the "Controls" menu in the visualizer, you will find sliders that you can use to drive the end-effector of the robot around. Give it a spin!

Example 1.2 (Teleoperation in 3D.)

I offered a 2D visualization / control first because everything is simpler in 2D. But the simulation is actually running fully in 3D. Run the second example to see it.

1.5 Model-based design and analysis

Thanks to progress in simulation, it is now possible to pursue a meaningful study of manipulation without needing a physical robot. But the software advances in simulation (of our robot dynamics, sensors, actuators and its environment) are not enough to support all of the topics in these notes. Manipulation research today leverages a myriad of advanced algorithms in perception, planning, and control. In addition to providing those individual algorithms, a major goal of these notes is to attempt to bridle the complexity of using them together in a systematic way.

Many of you will be familiar with ROS (the Robot Operating System). I believe that ROS was one of the best things to happen to robotics in the last decades. It meant that experts from different subdisciplines could easily share their expertise in the form of modular components. Components (as ROS packages) simply agree on the messages that they will send and receive on the network; packages can inter-operate even if they are written in different programming languages or even on different operating systems.

Although ROS makes it relatively easy to get started with manipulation, it doesn't serve my pedagogical goal of thinking clearly about manipulation. The modular approach to authoring the components is extremely good, and we will adopt it here. But in Drake we ask for a little bit more from each of the components -- essentially that they declare their states, parameters, and timing semantics in a consistent way -- so that we have a much better chance of understanding the complex relationships between systems. This has great practical value as well; the ability to debug a full manipulation stack with repeatable deterministic simulations (even if they include randomness) is surprisingly rare in the field but hugely valuable.

The key building block in our work will be Drake **Systems**, and systems can be combined in complex combinations into **Diagrams**. System diagrams have long been the modeling paradigm used in controls, and the software aspect of it will be very familiar to you if you've used tools like Simulink, LabView, or Modelica. These software tools refer to the block-diagram design paradigm as "model-based design".

Example 1.3 (System Diagrams)

Even the examples above, which relied on you to perform teleoperation instead of having an autonomy stack, were still the result of combining numerous parts. For any system (a system diagram is still a system) that you have in DRAKE, you can visualize that diagram directly in the notebook.

This graphic is interactive. Make sure you zoom in and click around to get a sense for the amount of complexity that we can abstract away in this framework. For instance, try expanding the `iiwa_controller` block.

Whenever you are ready to learn more about the Systems Framework in DRAKE, I recommend starting with the "Modeling Dynamical Systems" tutorial linked from the main Drake website.

Let me be transparent: not everybody likes this systems framework for manipulation. Some people are just trying to write code as fast as possible, and don't see the benefits of slowing down to declare their state variables, etc. It will likely feel like a burden to you the first time you go to write a new system. But it's precisely *because* we're trying to build complex systems quickly that I advocate this more rigorous approach. I believe that getting to the next level of maturity in our open-world manipulation systems requires more maturity from our building blocks.

1.6 ORGANIZATION OF THESE NOTES

The remaining chapters of these notes are organized around the component-level building blocks of manipulation. Many of these components each individually build on a wealth of literature (e.g. from computer vision, or dynamics and control). Rather than be overwhelmed, I've chosen to focus on delivering a consistent coherent presentation of the most relevant ideas from each field *as they relate to manipulation*, and pointers to more literature. Even finding a single notation across all of the fields can be a challenge!

The next few chapters will give you a minimal background on the relevant robot hardware that we are simulating, on (some of) the details about simulating them, and on the geometry and kinematics basics that we will use heavily through the notes.

For the remainder of the notes, rather than organize the chapters into "Perception", "Planning", and "Control", I've decided to spiral through these topics. In the first part, we'll do just enough perception, planning, and control to build up a basic manipulation system that can perform pick-and-place manipulation of known isolated objects. Then I'll try to motivate each new chapter by describing what our previous system cannot do, and what we hope it will do by the end of the chapter.

I welcome any feedback. And don't forget to interact!

1.7 EXERCISES

Exercise 1.1 (Familiarize yourself with Drake)

DRAKE is a powerful and mature software library that can support many advanced robotics applications. The motivation behind it is described in this blog post. It is extensively documented, but you have to know where to look for it.

1. Check out the growing list of DRAKE tutorials (linked from the main Drake page). In the `dynamical_systems` tutorial, to what value is the initial condition, $x(0)$, set when we simulate the `SimpleContinuousTimeSystem`?
2. The class-/function-level documentation is the most extensive documentation in Drake. When I'm working on Drake (in either C++ or Python), I most often have the C++ doxygen open. The Python documentation is (mostly) auto-generated from this and isn't curated as carefully; I tend to look there only in the rare cases that the Python interface differs from C++.
   In C++ doxygen, search for "Spatial Vectors". What ascii characters do we use to denote an angular acceleration in code?
3. Drake is open-source. There are no black-box algorithms here. If you ever want to see how a particular algorithm is implemented, or find examples of how to use a function, you can always look at the source code. These days you can use VS Code to explore the code right in your browser. What value of `convergence_tol` do I use in the unit test for "fitted value iteration"?

As you use Drake, if you have any questions, please consider asking them on stack**overflow** using the "drake" tag. The broader Drake developers community will often be able to answer faster (and/or better) than the course staff! And asking there helps to build a searchable knowledge base that will make Drake more useful and accessible for everyone.

REFERENCES

1. Matthew T Mason, "Toward robotic manipulation", *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 1, pp. 1--28, 2018.

2. Mortimer J Adler, "How to Mark a Book", *The Saturday Review of Literature*, July 6, 1941. [link]

Chapter 2
Let's get you a robot

In this chapter we're going to outfit your mech. I want to make sure you understand the robot hardware that we've selected for these notes, and how it compares to the other hardware available today. You should also come away with an understanding of how we simulate the robot and what commands you can send to the robot interface.

2.1 Robot description files

Although we are going to focus primarily on one particular set of hardware for the remainder of these notes, in this chapter I'll provide software examples with a number of different robots. One of the great things about modern robotics is that many of the tools we will develop over the course of these notes are quite general, and can be transferred from one robot to another easily. I could imagine a future version of these notes where you really do get to build out your robot in this chapter, and use your customized robot for the remaining chapters!

The ability to easily simulate/control a variety of robots is made possible in part by the proliferation of common file formats for describing our robots. Unfortunately, the field has not converged on a single preferred format (yet), and each of them have their quirks. Drake currently loads Universal Robot Description Format (URDF), Simulation Description Format (SDF), and has limited support for the MuJoCo format (MJCF). The Drake developers have been trying to upstream improvements to SDF rather than start yet another format, but we do have a very simple YAML specification called Drake Model Directives which makes it very quick and easy to load multiple robots/objects from these different file formats into one simulation; you saw an example of this in the .

2.2 Arms

There appear to be a lot of robotics arms available on the market. So how does one choose? Cost, reliability, usability, payload, range of motion, ...; there are many important considerations. And the choices we might make for a research lab might be very different than the choices we might make as a startup.

> **Example 2.1 (Robot arms)**
>
> I've put together a simple example to let you explore some of the various robot arms that are popular today. Let me know if your favorite arm isn't on the list yet!

There is one particular requirement which, if we want our robot to satisfy, quickly winnows the field to only a few viable platforms: that requirement is joint-torque sensing and control. Out of the torque-controlled robots on the market, I've chosen to use the Kuka LBR iiwa robot to use throughout these notes (I will try to use the lower case "iiwa" to be consistent with the manufacturer, but it looks wrong to me every time!).

Figure 2.1 - Kuka LBR iiwa robot. This one has a 7kg payload.

It's not absolutely clear that the joint-torque sensing and control feature is required, even for very advanced applications, but as a researcher who cares a great deal about the contact interactions between my robots and the world, I prefer to have the capability and explore whether I need it rather than wonder what life might have been like. To better understand why, let us start by understanding the difference between most robots, which are position-controlled, and the handful of robots that have accepted the additional cost and complexity to provide torque sensing and control.

2.2.1 Position-controlled robots

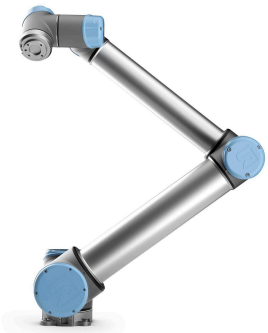


Figure 2.2 - Two popular position controlled manipulators. (Left) The UR10 from Universal Robotics. (Right)The ABB Yumi.

Most robot arms today are "position controlled" -- given a desired joint position (or joint trajectory), the robot executes it with relatively high precision. Basically all arms can be position controlled -- if the robot offers a torque control interface (with sufficiently high bandwidth) then we can certainly regulate positions, too. In practice, calling a robot "position controlled" is a polite way of saying that it does not offer torque control. Do you know why position control and not torque control is the norm?

Lightweight arms like the examples above are actuated with electric motors. For a reasonably high-quality electric motor (with windings designed to minimize torque ripple, etc), we expect the torque that the motor outputs to be directly proportional to the current that we apply:

$$\tau_{motor} = k_t i,$$

where τ_{motor} is the motor torque, i is the applied current, and k_t is the "motor torque constant". (Similarly, applied voltage has a simple (affine) relationship with the motor's steady-state velocity). If we can control the current, then why can we not control the torque?

The short answer is that to achieve reasonable cost and weight, we typically choose small electric motors with large gear reductions, and gear reductions come with a number of dynamic effects that are very difficult to model -- including backlash, vibration, and friction. So the simple relationship between current and torque breaks down. Conventional wisdom is that for large gear ratios (say $\gg 10$), the unmodeled terms are significant enough that they cannot be ignored, and torque is no longer simply related to current.

Position Control.

How can we overcome this challenge of not having a good model of the transmission dynamics? Regulating the current or speed *of the motor* only requires sensors on the motor side of the transmission. To accurately regulate the joint, we typically need add more sensors on the output side of the transmission. Importantly, although the torques due to the transmission are not known precisely, they are also not arbitrary -- for instance they will never add energy into the system. Most importantly, we can be confident that there is a *monotonically increasing* relationship between the current that we put into the motor and the torque at the joint, and ultimately the acceleration of the joint. Note that I chose the term monotonic carefully, meaning "non-decreasing" but *not* implying "strictly increasing", because, for instance, when a joint is starting from rest, static friction will resist small torques without having any acceleration at the output.

The most common sensor to add to the joint is a position sensor -- typically an encoder or potentiometer -- these are inexpensive, accurate, and robust. In practice, we think of these as providing (after some signal filtering/conditioning) accurate measurements of the joint position and joint velocity -- joint accelerations can also be obtained via differentiating twice but are generally considered more noisy and less suitable for use in tight feedback loops. Position sensors are sufficient for accurately tracking desired position trajectories of the arm. For each joint, if we denote the joint position as q and we are given a desired trajectory $q^d(t)$, then I can track this using proportional-integral-derivative (PID) control:

$$\tau = k_p(q^d - q) + k_d(\dot{q}^d - \dot{q}) + k_i \int (q^d - q) dt,$$

with k_p, k_d, and k_i being the position, velocity, and integral gains. PID control has a rich theory, and a trove of knowledge about how to choose the gains, which I will not reproduce here. I will note, however, that when we simulate position-controlled robots we often need to use different gains for the physical robot and for our simulations. This is due to the transmission dynamics, but also the fact that PID controllers in hardware typically output voltage commands (via pulse-width modulation) instead of current commands. Closing this modeling gap has traditionally not been a priority for robot simulation -- there are enough other details to get right which dominate the "sim-to-real" gap -- but I suspect that as the field matures the mainstream robotics simulators will eventually capture this, too.

Some of you are thinking, "I can train a neural network to model *anything*, I'm not afraid of difficult-to-model transmissions!" I do think there is reason to be optimistic about this approach; there are a number of initial demonstrations in this direction (e.g. [1]). This is not quite as useful as if we can have a first-principles model that can generalize to new actuators from a few parameters in a description file, but could be very productive.

An aside: link dynamics with a transmission.

One thing that might be surprising is that, despite the fact that the joint dynamics of a manipulator are highly coupled and state dependent, the PID gains are often chosen independently for each joint, and are constant (not gain-scheduled). Wouldn't you expect for the motor commands required for e.g. a robot arm at full extension holding a milk jug might be very different than the motor commands required when it is unloaded in a vertical hanging position? Surprisingly, the required gains/commands might not be as different as one would think.

Electric motors are most efficient at high speeds (often > 100 or 1,000 rotations per minute). We probably don't actually want our robots to move that fast even if they could! So nearly all electric robots have fairly substantial gear reductions, often on the order of 100:1; the transmission output turns one revolution for every 100 rotations of the motor, and the output torque is 100 times greater than the motor torque. For a gear ratio, n, actuating a joint q, we have

$$q_{motor} = nq, \quad \dot{q}_{motor} = n\dot{q}, \quad \ddot{q}_{motor} = n\ddot{q}, \qquad \tau_{motor} = \frac{1}{n}\tau.$$

Interestingly, this has a fairly profound impact on the resulting dynamics (given by $f = ma$), even for a single joint. Writing the relationship between joint torque and joint acceleration (no motors yet), we can write $ma = \sum f$ in the rotational coordinates as

$$I_{arm}\ddot{q} = \tau_{gravity} + \tau,$$

where I_{arm} is the moment of inertia. For example, for a simple pendulum, we might have

$$ml^2\ddot{q} = -mgl\sin q + \tau.$$

But the applied joint torque τ actually comes from the motor -- if we write this equation in terms of motor coordinates we get:

$$\frac{I_{arm}}{n}\ddot{q}_{motor} = \tau_{gravity} + n\tau_{motor}.$$

If we divide through by n, and take into account the fact that the motor itself has inertia (e.g. from the large spinning magnets) that is not affected by the gear ratio, then we obtain:

$$\left(I_{motor} + \frac{I_{arm}}{n^2}\right)\ddot{q}_{motor} = \frac{\tau_{gravity}}{n} + \tau_{motor}.$$

It's interesting to note that, even though the mass of the motors might make up only a small fraction of the total mass of the robot, for highly geared robots they can play a significant role in the dynamics of the joints. We use the term *reflected inertia* to denote the inertial load that is felt on the opposite side of a transmission, due to the scaling effect of the transmission. The "reflected inertia" of the arm at the motor is cut by the square of the gear ratio; or the "reflected inertia" of the motor at the arm is multiplied by the square of the gear ratio. This has interesting consequences -- as we move to the multi-link case, we will see that I_{arm} is a state-dependent function that captures the inertia of the actuated link and also the inertial coupling of the other joints in the manipulator. I_{motor}, on the other hand, is constant and only effects the local joint. For large gear ratios, the I_{motor} terms dominate the other terms, which has two important effects: 1) it effectively diagonalizes the manipulator equations (the inertial coupling terms are relatively small), and 2) the dynamics are relatively constant throughout the workspace (the state-dependent terms are relatively small). These effects make it relatively easy to tune constant feedback gains for each joint individually that perform well in all configurations.

2.2.2 Torque-controlled robots

Although not as common, there are a number of robots that do support direct control of the joint torques. There are a handful of ways that this capability can be realized.

It *is* possible to actuate a robot using electric motors that require only a small gear reduction (e.g. $\leq$ 10:1) where the frictional forces are negligible. In the past, these "direct-drive robots"[2] had enormous motors and limited payloads. More recently, robots like the Barrett WAM arm used cable drives to keep the arm light but having large motors in the base. And just in the last few years, we've seen progress in high-torque outrunner and frameless motors bringing in a new generation of low-cost, "quasi-direct-drive" robots: e.g. MIT Cheetah [3], Berkeley Blue, and Halodi Eve.

Hydraulic actuators provide another solution for generating large torques without large transmissions. Sarcos had a series of torque-controlled arms (and humanoids), and many of the most famous robots from Boston Dynamics are based on hydraulics (though there is an increasing trend towards electric motors). These robots typically have a single central pump and each actuator has a (lightweight) valve that can shunt fluid through the actuator or through a bypass; the differential pressure across the actuator is at least approximately related to the resulting force/torque.

Another approach to torque control is to keep the large gear-ratio motors, but add sensors to directly measure the torque at the joint side of the actuator. This is the approach used by the Kuka iiwa robot that we use in the example throughout this text; the iiwa actuators have strain gauges integrated into the transmission. However there is a trade-off between the stiffness of the transmission and the accuracy of the force/torque measurement [4] -- the iiwa transmission includes an explicit "Flex Spline" with a stiffness around 5000 Nm/rad [5]. Taking this idea to an extreme, Gill Pratt proposed "series-elastic actuators" that have even lower stiffness springs in the transmission, and proposed measuring joint position on both the motor and joint sides of the transmission to estimate the applied torques [6]. For example, the Baxter and Sawyer robots from Rethink used series-elastic actuators; I don't think they ever published the spring stiffness values but similarly-motivated series-elastic actuators from HEBI robotics are closer to 100 Nm/rad. Even for the iiwa actuators, the joint elasticity is significant enough that the low-level controllers go to great length to take it into account explicitly in order to achieve high-performance control of the joints[7]. We will discuss these details when we get to the chapter covering force control.

2.2.3 A proliferation of hardware

The low-cost torque-controlled arms that I mentioned above are just the beginnings in what promises to be a massive proliferation of robotic arms. During the pandemic, I saw a number of people using inexpensive robots like the xArm at home. As demand increases, costs will continue to come down.

Let me just say that, compared to working on legged robots, where for decades we did our research on laboratory prototypes built by graduate students (and occasionally professors!) in the machine shop down the hall, the availability of professionally engineered, high-quality, high-uptime hardware is an absolute treat. This also means that we can test algorithms in one lab and have another lab perhaps at another university testing algorithms on almost identical hardware; this facilitates levels of repeatability and sharing that were impossible before. The fact that the prices are coming down, which will mean many more similar robots in many more labs/environments, is one of the big reasons why I am so optimistic about the next few years in the field.

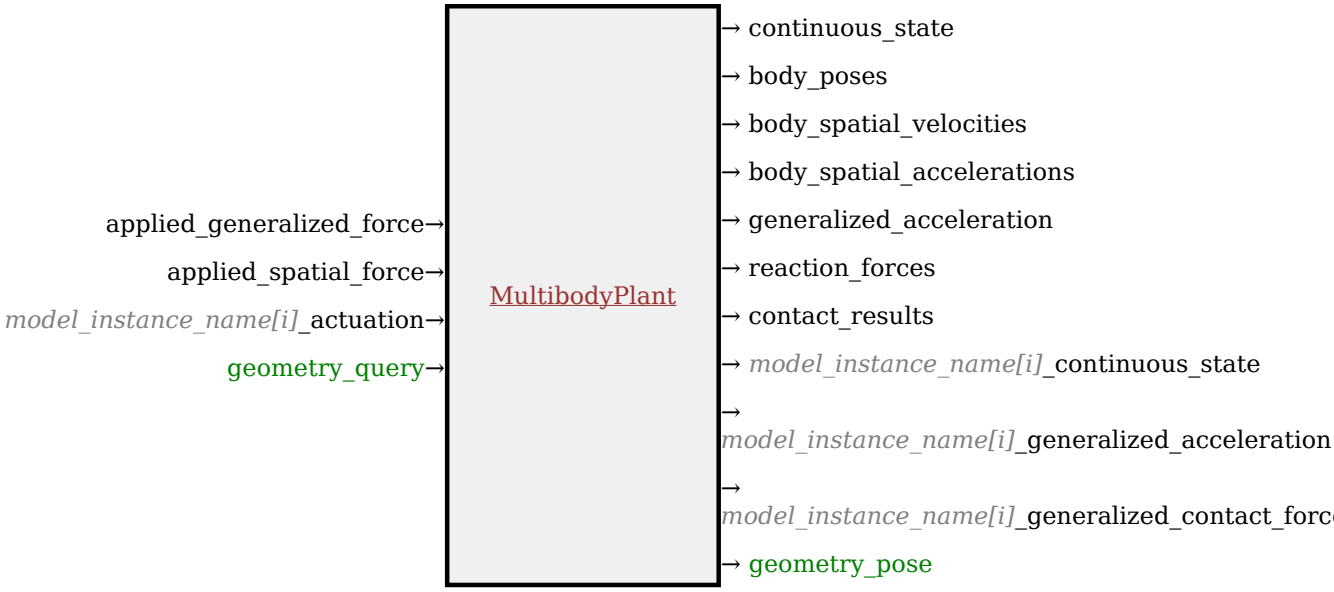
It's a good time to be working on manipulation!

2.2.4 Simulating the Kuka iiwa

It's time to simulate our chosen robotic arm. The first step is to obtain a robot description file (typically URDF or SDF). For convenience, we ship the models for a few robots, including iiwa, with Drake. If you're interested in simulating a different robot, you can find either a URDF or SDF describing most commercial robots somewhere online. But a word of warning: the quality of these models can vary wildly. We've seen surprising errors in even the kinematics (link lengths, geometries, etc), but the dynamics properties (inertia, friction, etc) in particular are often not accurate at all. Sometimes they are not even mathematically consistent (e.g. it is possible to specify an inertial matrix in URDF/SDF which is not physically realizable by any rigid body). Drake will complain if you ask it to load a file with this sort of violation; we would rather alert you early than start generating bogus simulations. There is also increasingly good support for exporting to a robot format directly from CAD software like Solidworks.

Now we have to import this robot description file into our physics engine. In Drake, the physics engine is called `MultibodyPlant`. The term "plant" may seem odd but it is pervasive; it is the word used in the controls literature to represent a physical system to be controlled, which originated in the control of chemical plants. This connection to control theory is very important to me. Not many physics engines in the world go to the lengths that Drake does to make the physics engine compatible with control-theoretic design and analysis.

The `MultibodyPlant` has a class interface with a rich library of methods to work with the kinematics and dynamics of the robot. If you need to compute the location of the center of mass, or a kinematic Jacobian, or any similar queries, then you'll be using this class interface. A `MultibodyPlant` also implements the interface to be used as a `System`, with input and output ports, in Drake's systems framework. In order to simulate, or analyze, the combination of a `MulitbodyPlant` with other systems like our perception, planning, and control systems, we will be assembling block diagrams.



As you might expect for something as complex and general as a physics engine, it has many input and output ports; most of them are optional. I'll illustrate the mechanics of using these in the following example.

Example 2.2 (Simulating the passive iiwa)

It's worth spending a few minutes with this example, which should help you understand not only the physics engine, but some of the basic mechanics of working with simulations in Drake.

The best way to visualize the results of a physics engine is with a 2D or 3D visualizer. For that, we need to add the system which curates the geometry of a scene; in Drake we call it the `SceneGraph`. Once we have a `SceneGraph`, then there are a number of different visualizers and sensors that we can add to the system to actually render the scene.

Example 2.3 (Visualizing the scene)

This example is far more interesting to watch. Now we have the 3D visualization!

You might wonder why `MultibodyPlant` doesn't handle the geometry of the scene as well. Well, there are many applications in which we'd like to render complex scenes, and use complex sensors, but supply custom dynamics instead of using the default physics engine. Autonomous driving is a great example; in that case we want to populate a `SceneGraph` with all of the geometry of the vehicles and environment, but we often want to simulate the vehicles with very simple vehicle models that stop well short of adding tire mechanics into our physics engine. We also have a number of examples of this workflow in my Underactuated Robotics course, where we make extensive use of "simple models".

We now have a basic simulation of the iiwa, but already some subtleties emerge. The physics engine needs to be told what torques to apply at the joints. In our example, we apply zero torque, and the robot falls down. In reality, that never happens; in fact there is essentially never a situation where the physical iiwa robot experiences zero torque at the joints, even when the controller is turned off. Like many mature industrial robot arms, iiwa has mechanical brakes at each joint that are engaged whenever the controller is turned off. To simulate the robot with the controller turned off, we would need to tell our physics engine about the torques produced by these brakes.

In fact, even when the controller is turned on, and despite the fact that it is a torque-controlled robot, we can never actually send zero torques to the motors. The iiwa software interface accepts "feed-forward torque" commands, but it will always add these as additional torques to its low-level controller which is compensating for gravity and the motor/transmission mechanics. This often feels frustrating, but probably we don't actually want to get into the details of simulating the drive mechanics.

As a result, the simplest reasonable simulation we can provide of the iiwa must include a simulation of Kuka's low-level controller. We will use the iiwa's "joint impedance control" mode, and will describe the details of that once they become important for getting the robot to perform better. For now, we can treat it as given, and produce our simplest reasonable iiwa simulation.

Example 2.4 (Adding the iiwa low-level controller)

This example adds the iiwa controller and sets the desired *positions* (no longer the desired torques) to be the current state of the robot. It's a more faithful simulation of the real robot. I'm sorry that it is boring once again!

As a final note, you might think that simulating the *dynamics* of the robot is overkill, if our only goal is to simulate manipulation tasks where the robot is moving only relatively slowly, and effects of mass, inertia and forces might be less important than just the positions that the robot (and the objects) occupy in space. I would actually agree with you. But it's surprisingly tricky to get a *kinematic* simulation to respect the basic rules of interaction; e.g. to know when the object gets picked up or when it does not (see, for instance [8]). Currently, in Drake, we mostly use the full physics engine for simulation, but often use simpler models for manipulation planning and control.

2.3 Hands

You might have noticed that the iiwa model does not actually have a hand attached; the robot ships with a mounting plate so that you can attach the "end-effector" of

your choice (and some options on access ports so you can connect your end-effector to the computer without wires running down the outside of the robot). So now we have another decision to make: what hand should we use?

Example 2.5 (Robot hands)

We can explore different hand models in Drake using the same sort of interface we used for the arms, though I don't have as many hand models here yet. Let me know if your favorite hand isn't on the list!

It is interesting that, when it comes to robot end effectors, researchers in manipulation tend to partition themselves into a few distinct camps.

2.3.1 Dexterous hands

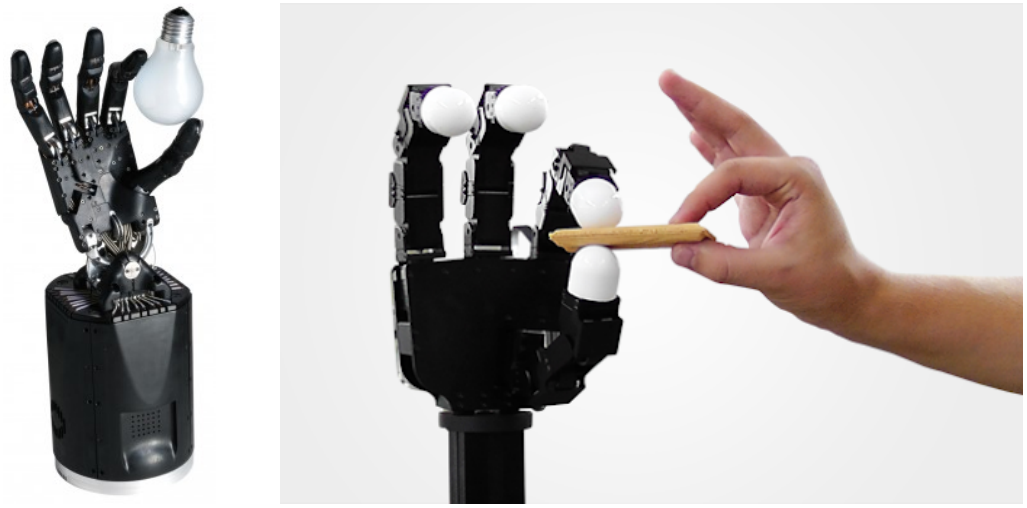


Figure 2.3 - Dexterous hands. Left: the Shadow Dexterous Hand. Right: the Allegro Hand.

Of course, our fascination with the human hand is well placed, and we dream of building robotic hands that are as dexterous and sensor-rich. But the reality is that we aren't there yet. Some people choose to pursue this dream and work with the best dexterous hands on the market, and struggle with the complexity and lack of robustness that ensues. The famous "learning dexterity" project from OpenAI used the Shadow hand for playing with a Rubik's cube, and the work that had to go into the hand in order to support the endurance learning experiments was definitely a part of the story. There is a chance that new manufacturing techniques could really disrupt this space -- videos like this one of FLLEX v2 look amazing[9] -- and I am very optimistic that we'll have more capable and robust dexterous hands in the not-so-distant future.

2.3.2 Simple grippers

Click here to watch the video.

Figure 2.4 - This video of tele-operation with the PR1 from Ken Salisbury's group is now a classic example of doing amazingly useful things with a very simple hand. Check out their website for more videos, including sweeping, fetching a beer, and unloading a dishwasher.

Another camp points out that dexterous hands are not necessary -- I can give you a simple gripper from the toy store and you can still accomplish amazingly useful tasks around the home. The PR1 videos above are a great demonstration of this.

Another important argument in favor of simple hands is the elegance and clarity that comes from reducing the complexity. If thinking clearly about simple grippers helps us understand more deeply *why* we need more dexterous hands (I think it will), then great. For most of these notes, a simple two-fingered gripper will serve our pedagogical goals the best. In particular, I've selected the Schunk WSG 050, which we have used extensively in our research over the last few years. We'll also explore a few different end-effectors in later chapters, when they help to explain the concepts.

To be clear: just because a hand is simple (few degrees of freedom) does not mean that it is low quality. On the contrary, the Schunk WSG is a very high-quality gripper with force control and force measurement at its single degree of freedom that surpasses the fidelity of the Kuka. It would be hard to achieve the same in a dexterous hand with many joints.

2.3.3 Soft/underactuated hands

Finally, the third and newest camp is promoting clever mechanical designs for hands, which are often called "underactuated hands". The basic idea is that, for many tasks, you might not need as many actuators in your hand has you have joints. Many underactuated hands use a cable-drive mechanism to close the fingers, where a single tendon can cause multiple joints in the finger to bend. When designed correctly, these mechanisms can allow the finger to conform passively to the shape of an object being grasped with no change in the actuator command (c.f. [10]). Cables are not required for this concept to work; qualitatively similar behavoir can be achieved using clever rigid mechanical linkages, as well.

Figure 2.5 - Underactuated hands. Left: the RightHand Robotics Reflex2 is a descendant of the i-HY hand[10]. Right: the Robotiq 3-fingered gripper.

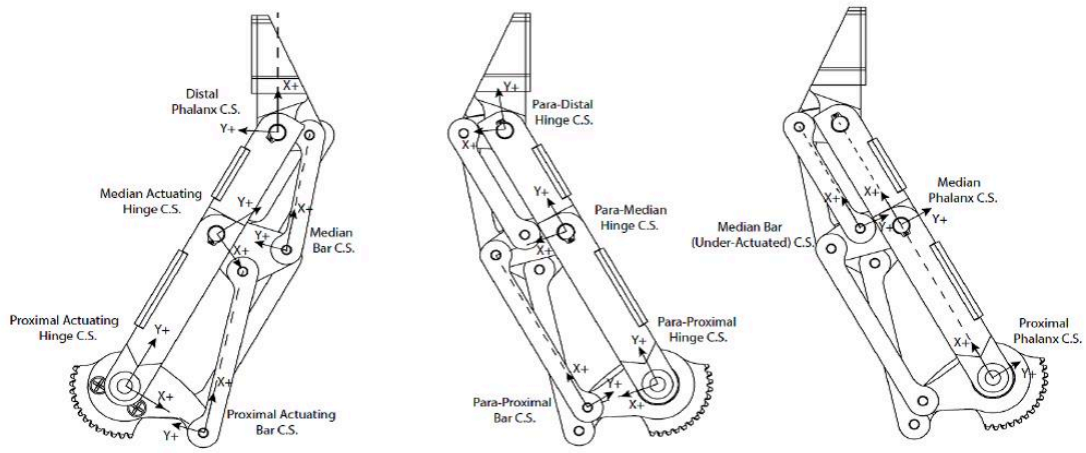


Figure 2.6 - A clever mechanical linkage allows the underactuated Robotiq 3-fingered gripper to comply to an object being grasped.

Taking the idea of underactuation and passive compliance to an extreme, recent years have also seen a number of hands (or at least fingers) that are completely soft. The "soft robotics community" is rapidly changing the state of the art in terms of robot fabrication, with appendages, actuators, sensors, and even power sources that can be completely soft. These technologies promise to improve durability, decrease cost, and potentially be more safe for operating around people.

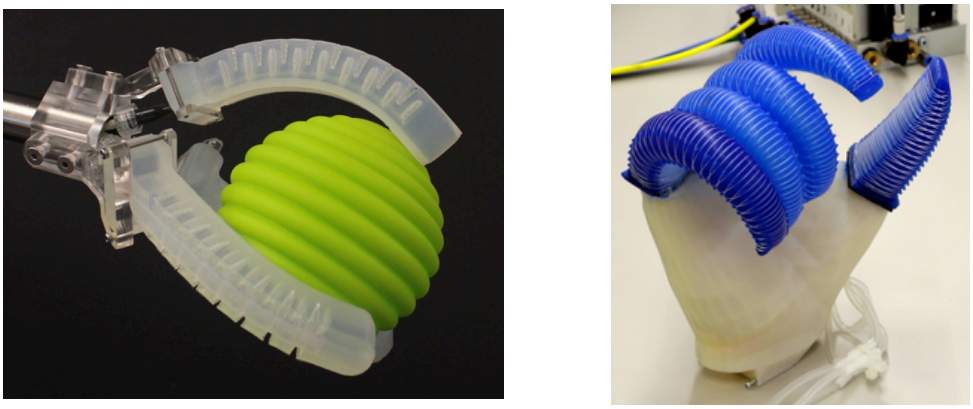


Figure 2.7 - Underactuated hands. Left: A 3D-printed soft hand from Harvard (Image credit: Ryan Truby). Right: The RBO Hand 2 (Image credit: Disney Research Zurich).

Underactuated hands can be excellent examples of mechanical design reducing the burden on the actuators / control system. Often these hands are amazingly good at some range of tasks (most often "enveloping grasps"), but not as general purpose. It would be very hard to use one of these to, for instance, button my shirt. They are, however, becoming more and more dexterous; check out the video below!

Click here to watch the video.

2.3.4 Other end effectors

Not all end effectors need to operate like dexterous or simplified human hands. Many industrial applications these days are doing a form of pick and place manipulation using vacuum grippers (also known as suction-cup grippers). Suction cups work extremely well on many but not all objects. Some objects are too soft or porous to be suctioned effectively. Some objects are too fragile or heavy to be lifted from a vacuum at the top of the object, and must be supported from below. Some hands have suction in the palms to achieve an initial pick, but still use more

traditional fingers to stabilize a grasp.

There are numerous other clever gripper technologies. One of my favorites is the jamming gripper. These grippers are made of a balloon filled with coffee grounds, or some other granular media; pushing down the balloon around an object allows the granular media to flow around the object, but then applying a vacuum to the balloon causes the granular media to "jam", quickly hardening around the object to make a stable grasp [11].

Click here to watch the video.

Here is another clever design with actuated rollers at the finger tips to help with in-hand reorientation.

Finally, a reasonable argument against dexterous hands is that even humans often do some of their most interesting manipulation not with the hand directly, but through tools. I particularly liked the response that Matt Mason, one of the main advocates for simple grippers throughout the years, gave to a question at the end of one of our robotics seminars: he argued that useful robots in e.g. the kitchen will probably have special purpose tools that can be changed quickly. In applications where the primary job of the dexterous hand is to change tools, we might skip the complexity by mounting a "tool changer" directly to the robot and using tool-changer-compatible tools.

2.3.5 If you haven't seen it...

One time I was attending an event where the registration form asked us "what is your favorite robot of all time, real or fictional". That is a tough question for someone who loves robots! But the answer I gave was a super cool "high-speed multifingered hand" by the Ishikawa group; a project that started turning out amazing results back in 2004! They "overclocked" the hand -- sending more current for short durations than would be reasonable for any longer applications -- and also used high-speed cameras to achieve these results. And they had a Rubik's cube demo, too, in 2017.

Click here to watch the video.

So good!

2.4 Sensors

I haven't said much yet about sensors. In fact, sensors are going to be a major topic for us when we get to perception with (depth) cameras, and when we think about tactile sensing. But I will defer those topics until we need them.

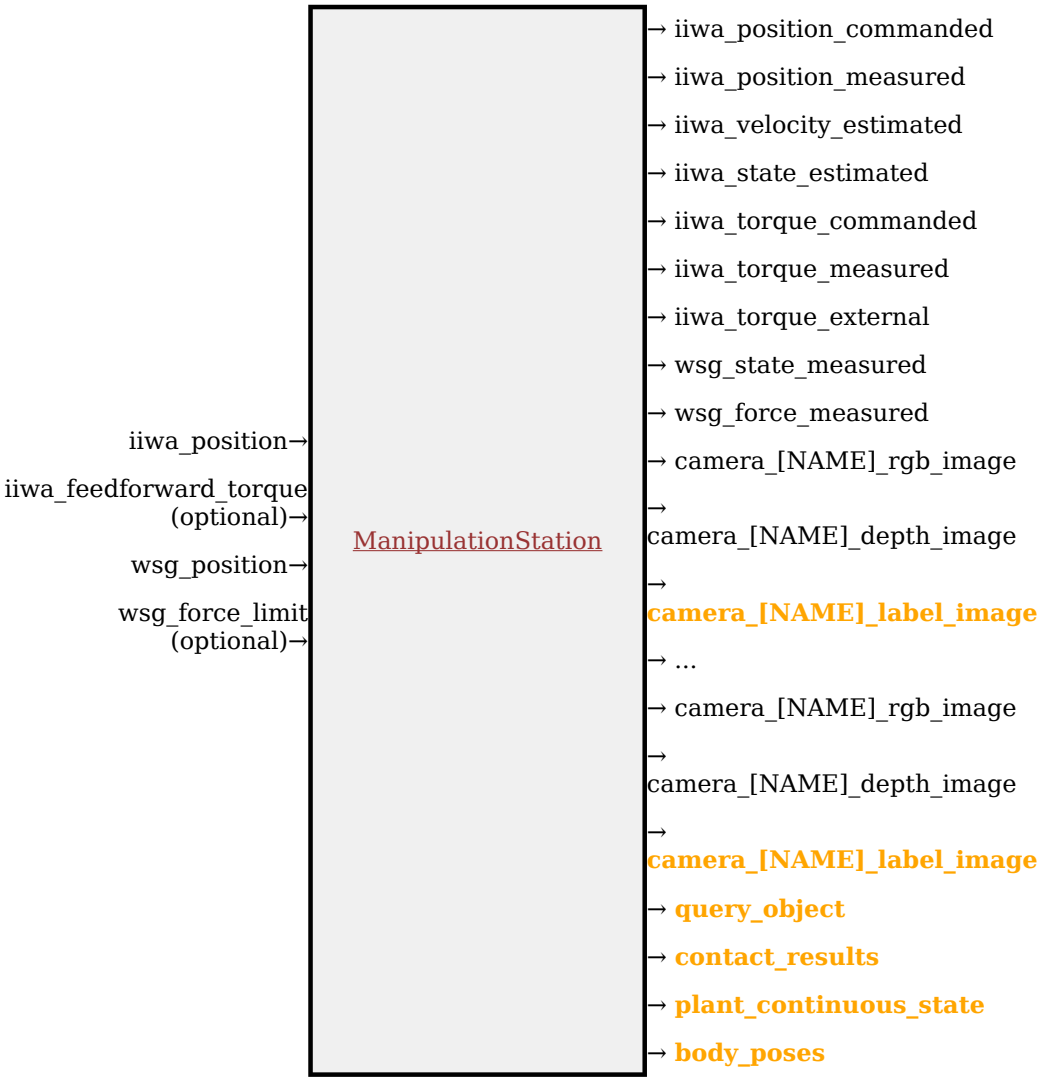
For now, let us focus on the joint sensors on the robot. Both the iiwa and the Schunk WSG provide joint feedback -- the iiwa driver gives "measured position", "estimated velocity", and "measured torque" at each of its seven joints; remember that joint accelerations are typically considered too noisy to rely on. Similarly the Schunk WSG outputs "measured state" (position + velocity) and "measured force". We can make all of these available as ports in a block diagram.

2.5 Putting it all together

If you've worked through the examples, you've seen that a proper simulation of our robot is more than just a physics engine -- it requires assembling physics, actuator and sensor models, and low-level robot controllers into a common framework. In practice, in Drake, that means that we are assembling increasingly sophisticated block diagrams.

One of the best things about the block-diagram modeling paradigm is the power

of abstraction and encapsulation. We can assemble a `Diagram` that contains all of the components necessary to simulate our hardware platform and its environment, which we will refer to affectionately as the "Manipulation Station". All together, the `ManipulationStation` system looks like this:



Example 2.6 (Manipulation station in the teleop demo)

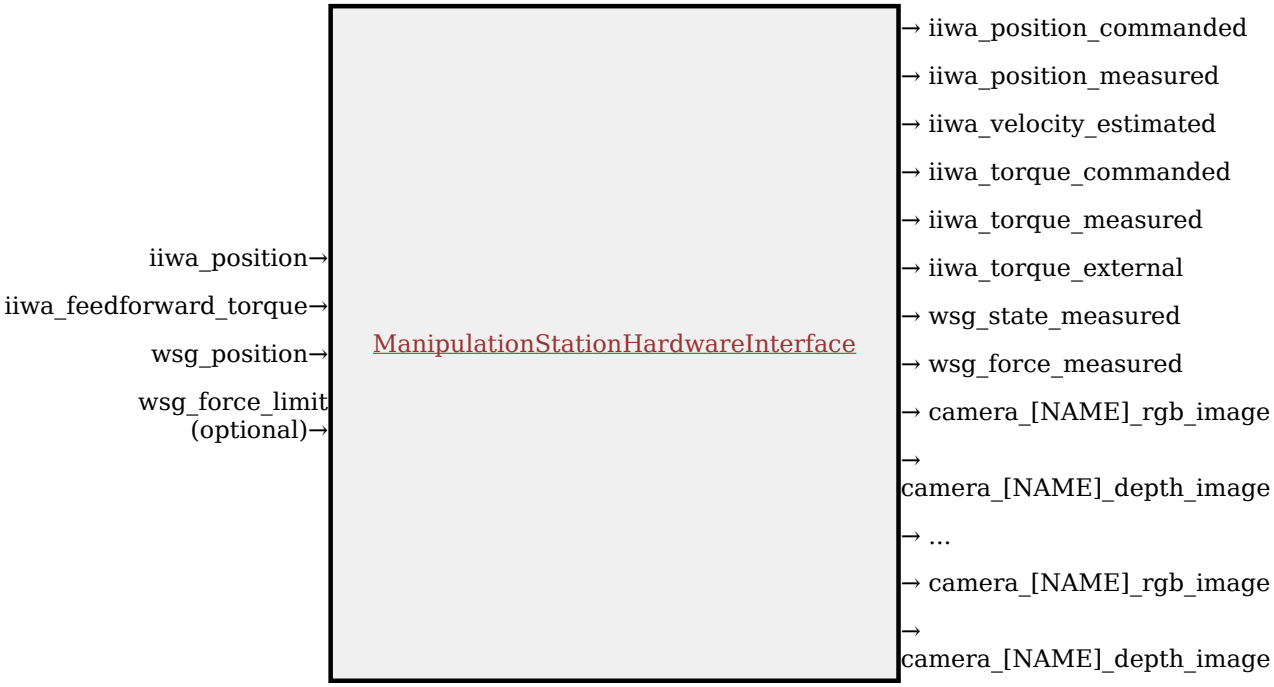
The teleop notebook in the first chapter used the `MakeManipulationStation` interface to set up the simulation. Now you have a better sense for what is going on inside that subsystem! Here is the link if you want to take a look at that example again:

Example 2.7 (A bimanual manipulation station)

I've intentionally provided this version of the `MakeManipulationStation` method in Python so that you can easily customize it to your needs. But even the version I've provided should address many use cases. Here is a simple example of setting up a bimanual version of the manipulation station:

This diagram itself can then be used as a `System` in additional diagrams, which can include our perception, planning, and, higher-level control systems. This model

also defines the abstraction between the simulation and the real hardware. We offer an almost identical system, the `ManipulationStationHardwareInterface`. If you replace this directly in place of the `ManipulationStation`, then the same code you've developed in simulation can be run directly on the real robot. The ports that are available only in simulation, but not in reality, are colored **orange** on the `ManipulationStation` system.



The `ManipulationStationHardwareInterface` is also a diagram, but rather than being made up of the simulation components like `MultibodyPlant` and `SceneGraph`, it is made up of systems that perform network message passing to interface with the small executables that talk to the individual hardware drivers. If you dig under the covers, you will see that we use LCM for this instead of ROS messages, precisely because LCM is a lighter-weight dependency for our public repository. But many Drake developers/users use Drake in a ROS/ROS2 ecosystem.

If you do have your own similar robot hardware available, and want to run the hardware interface on your machines, I've started putting together a list of drivers and bill of materials in the appendix.

Example 2.8 (The iiwa with an Allegro hand)

If you want to set up a simulation and controller with entirely different hardware, then I hope you will be able to look at the `MakeManipulationStation` code and adapt it to your needs. For instance, here is a simple example of simulating the iiwa with the Allegro hand attached instead of the Schunk WSG gripper.

2.6 Exercises

Exercise 2.1 (Role of Reflected Inertia)

For this exercise you will investigate the effect of reflected inertia on the joint-space dynamics of the robot, and how it affects simple position control laws. You will work exclusively in . You will be asked to complete the following steps:

a. Derive the first-order state-space dynamics $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ of a simple pendulum with a motor and gearbox.
b. Compare the behavior of the direct-driven simple pendulum and the simple pendulum with a high-ratio gearbox, under the same position control law.

Exercise 2.2 (Input and Output Ports on the Manipulation Station)

For this exercise you will investigate how a manipulation station is abstracted in Drake's system-level framework. You will work exclusively in . You will be asked to complete the following steps:

a. Learn how to probe into inputs and output ports of the manipulation station and evaluate their contents.
b. Explore what different ports correspond to by probing their values.

Exercise 2.3 (Direct Joint Teleop in Drake)

For this exercise you will implement a method for controlling the joints of a robot in Drake. You will work exclusively in , and should use the as a reference. You will be asked to complete the following steps:

a. Replace the teleop interface in the chapter 1 example with different Drake functions that allow for directly controlling the joints of the robot.

References

1. Jemin Hwangbo and Joonho Lee and Alexey Dosovitskiy and Dario Bellicoso and Vassilios Tsounis and Vladlen Koltun and Marco Hutter, "Learning agile and dynamic motor skills for legged robots", *Science Robotics*, vol. 4, no. 26, pp. eaau5872, 2019.

2. Haruhiko Asada and Kamal Youcef-Toumi, "Direct-Drive Robots - Theory and Practice", The MIT Press , 1987.

3. P. M. Wensing and A. {Wang} and S. {Seok} and D. {Otten} and J. {Lang} and S. {Kim}, "Proprioceptive Actuator Design in the MIT Cheetah: Impact Mitigation and High-Bandwidth Physical Interaction for Dynamic Legged Robots", *IEEE Transactions on Robotics*, vol. 33, no. 3, pp. 509-522, June, 2017.

4. Navvab Kashiri and Jörn Malzahn and Nikos Tsagarakis, "On the Sensor Design of Torque Controlled Actuators: A Comparison Study of Strain Gauge and Encoder Based Principles", *IEEE Robotics and Automation Letters*, vol. PP, 02, 2017.

5. A Wedler and M Chalon and K Landzettel and M G{\"o}rner and E Kr{\"a}mer and R Gruber and A Beyer and HJ Sedlmayr and B Willberg and W Bertleff and others, "DLRs dynamic actuator modules for robotic space applications", *Proceedings of the 41st Aerospace Mechanisms Symposium* , May 16-18, 2012.

6. G. A. Pratt and M. M. {Williamson}, "Series elastic actuators", *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots* , vol. 1, pp. 399-406

vol.1, Aug, 1995.

7. Alin Albu-Schaffer and Christian Ott and Gerd Hirzinger, "A unified passivity-based control framework for position, torque and impedance control of flexible joint robots", *The international journal of robotics research*, vol. 26, no. 1, pp. 23--39, 2007.

8. Tao Pang and Russ Tedrake, "A Robust Time-Stepping Scheme for Quasistatic Rigid Multibody Systems", *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* , 2018. [link]

9. Yong-Jae Kim and Junsuk Yoon and Young-Woo Sim, "Fluid Lubricated Dexterous Finger Mechanism for Human-Like Impact Absorbing Capability", *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3971--3978, 2019.

10. Lael U. Odhner and Leif P. Jentoft and Mark R. Claffee and Nicholas Corson and Yaroslav Tenzer and Raymond R. Ma and Martin Buehler and Robert Kohout and Robert D. Howe and Aaron M. Dollar, "A Compliant, Underactuated Hand for Robust Manipulation", *International Journal of Robotics Research (IJRR)*, vol. 33, no. 5, pp. 736-752, 2014.

11. Eric Brown and Nicholas Rodenberg and John Amend and Annan Mozeika and Erik Steltz and Mitchell R Zakin and Hod Lipson and Heinrich M Jaeger, "Universal robotic gripper based on the jamming of granular material", *Proceedings of the National Academy of Sciences*, vol. 107, no. 44, pp. 18809–18814, 2010.

Chapter 3
Basic Pick and Place

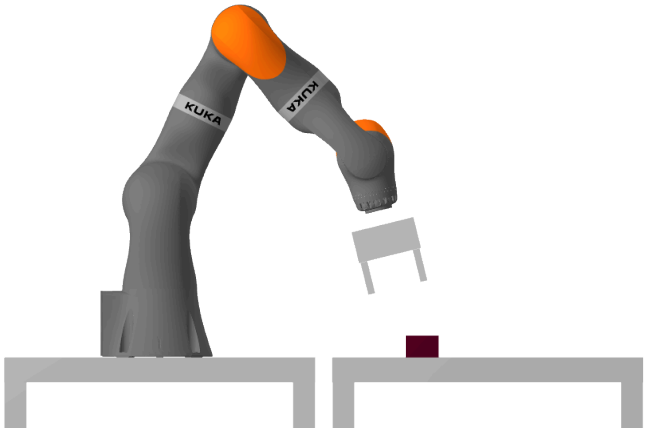


Figure 3.1 - Your challenge: command the robot to pick up the brick and place it in a desired position/orientation.

The stage is set. You have your robot. I have a little red foam brick. I'm going to put it on the table in front of your robot, and your goal is to move it to a desired position/orientation on the table. I want to defer the *perception* problem for one more chapter, and will let you assume that you have access to a perfect measurement of the current position/orientation of the brick. Even without perception, completing this task requires us to build up a basic toolkit for geometry and kinematics; it's a natural place to start.

First, we will establish some terminology and notation for kinematics. This is one area where careful notation can yield dividends, and sloppy notation will inevitably lead to confusion and bugs. The Drake developers have gone to great length to establish and document a consistent multibody notation, which we call "Monogram Notation". The documentation even includes some of the motivation/philosophy behind that notation. I'll use the monogram notation throughout this text.

If you'd like a more extensive background on kinematics than what I provide here, my favorite reference is still [1]. For free online resources, Chapters 2 and 3 of the 1994 book by Murray et al. (now free online)[2] are also excellent, as are the first seven chapters of Modern Robotics by Lynch and Park[3] (they also have excellent accompanying videos). Unfortunately, with three different references you'll get three (slightly) different notations; ours is most similar to [1]. The monogram notation is developed in some detail in [4].

Please don't get overwhelmed by how much background material there is to know! I am personally of the opinion that a clear understanding of just a few basic ideas should make you very effective here. The details will come later, if you need them.

3.1 Monogram Notation

The following concepts are disarmingly subtle. I've seen incredibly smart people assume they knew them and then perpetually stumble over notation. I did it for years myself. Take a minute to read this carefully!

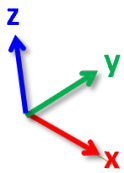
Perhaps the most fundamental concept in geometry is the concept of a point. Points occupy a position in space, and they can have names, e.g. point A, C, or more descriptive names like B_{cm} for the center of mass of body B. We'll denote the position of the point by using a position vector p^A; that's p for position, and not for point,

because other geometric quantities can also have a position.

But let's be more careful. Position is actually a relative quantity. Really, we should only ever write the position of two points relative to each other. We'll use e.g. ${}^{A}p^{C}$ to denote the position of C measured from A. The left superscript looks mighty strange, but we'll see that it pays off once we start transforming points.

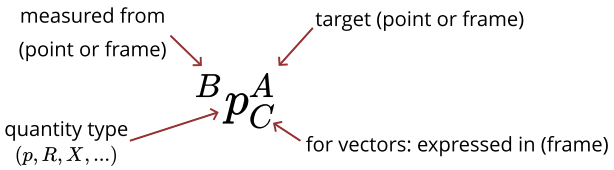
Every time we describe the (relative) position as a vector of numbers, we need to be explicit about the frame we are using, specifically the "expressed-in" frame. All of our frames are defined by orthogonal unit vectors that follow the "right-hand rule". We'll give a frame a name, too, like F. If I want to write the position of point C measured from point A, expressed in frame F, I will write ${}^{A}p^{C}_{F}$. If I ever want to get just a single component of that vector, e.g. the x component, then I'll use ${}^{A}p^{C}_{F_x}$. In some sense, the "expressed-in" frame is an implementation detail; it is only required once we want to represent the multibody quantity as a vector (e.g. in the computer).

That is seriously heavy notation. I don't love it myself, but it's the most durable I've got, and we'll have shorthand for when the context is clear.



There are a few very special frames. We use W to denote the *world frame*. We think about the world frame in Drake using vehicle coordinates (positive x to the front, positive y to the *left*, and positive z is up). The other particularly special frames are the *body frames*: every body in the multibody system has a unique frame attached to it. We'll typically use B_i to denote the frame for body i.

Frames have a position, too -- it coincides with the frame origin. So it is perfectly valid to write ${}^{W}p^{A}_{W}$ to denote the position of point A measured from the origin of the world frame, expressed in the world frame. Here is where the shorthand comes in. If the position of a quantity is measured from a frame, and expressed in the same frame, then we can safely omit the subscript. ${}^{F}p^{A} \equiv {}^{F}p^{A}_{F}$. Furthermore, if the "measured from" field is omitted, then we assume that the point is measured from W, so $p^{A} \equiv {}^{W}p^{A}_{W}$.



Frames also have an orientation. We'll use R to denote a *rotation*, and follow the same notation, writing ${}^{B}R^{A}$ to denote the rotation of frame A measured from frame B. Unlike vectors, pure rotations do not have an additional "expressed in" frame.

A frame F can be specified completely by a position and rotation measured from another frame. Taken together, we call the position and rotation a *spatial pose*, or just *pose*. A *spatial transform*, or just *transform*, is the "verb form" of pose. In Drake we use `RigidTransform` to represent a pose/transform, and denote it with the letter X. ${}^{B}X^{A}$ is the pose of frame A measured from frame B. When we talk about the pose of an object O, without mentioning a reference frame explicitly, we mean ${}^{W}X^{O}$ where O is the body frame of the object. We do not use the "expressed in" frame subscript for pose; we always want the pose expressed in the reference frame.

The Drake documentation also discusses how to use this notation in code. In short, ${}^{B}p^{A}_{C}$ is written `p_BA_C`, ${}^{B}R^{A}$ as `R_BA`, and ${}^{B}X^{A}$ as `X_BA`. It works, I promise.

3.2 Pick and Place via Spatial Transforms

Now that we have the notation, we can formulate our approach to the basic pick and place problem. Let us call our object, O, and our gripper, G. Our idealized perception

sensor tells us ${}^{W}X^{O}$. Let's create a frame O_d to describe the "desired" pose of the object, ${}^{W}X^{O_d}$. So pick and place manipulation is simply trying to make $X^{O} = X^{O_d}$.

To accomplish this, we will assume that the object doesn't move relative to the world (${}^{W}X^{O}$ is constant) when the gripper is open, and the object doesn't move relative to the gripper (${}^{G}X^{O}$ is constant) when the gripper is closed. Then we can:

- move the gripper in the world, X^G, to an appropriate pose measured from the object: ${}^{O}X^{G_{grasp}}$.
- close the gripper.
- move the gripper+object to the desired pose, $X^{O} = X^{O_d}$.
- open the gripper, and retract the hand.

There is just one more important detail: to approach the object without colliding with it, we will insert a "pregrasp pose", ${}^{O}X^{G_{pregrasp}}$, above the object as an intermediate step. We'll use the same transform to retract away from the object when we set it down.

Clearly, programming this strategy requires good tools for working with these transforms, and for relating the pose of the gripper to the joint angles of the robot.

3.3 Spatial Algebra

Here is where we start to see the pay-off from our heavy notation, as we define the rules of converting positions, rotations, poses, etc. between different frames. Without the notation, this invariably involves me with my right hand in the air making the "right-hand rule", and my head twisting around in space. With the notation, it's a simple matter of lining up the symbols properly, and we're more likely to get the right answer!

Here are the basic rules of algebra for our spatial quantities:

- Positions expressed in the same frame can be added when their reference and target symbols match:

$$ {}^{A}p^{B}_{F} + {}^{B}p^{C}_{F} = {}^{A}p^{C}_{F}. \tag{1} $$

  Addition is commutative, and the additive inverse is well defined:

$$ {}^{A}p^{B}_{F} = -{}^{B}p^{A}_{F}. \tag{2} $$

  Those should be pretty intuitive; make sure you confirm them for yourself.
- Multiplication by a rotation can be used to change the "expressed in" frame:

$$ {}^{A}p^{B}_{G} = {}^{G}R^{F}\,{}^{A}p^{B}_{F}. \tag{3} $$

  You might be surprised that a rotation alone is enough to change the expressed-in frame, but it's true. The position of the expressed-in frame does *not* affect the relative position between two points.
- Rotations can be multiplied when their reference and target symbols match:

$$ {}^{A}R^{B}\,{}^{B}R^{C} = {}^{A}R^{C}. \tag{4} $$

  The inverse operation is also simply defined:

$$ \left[{}^{A}R^{B}\right]^{-1} = {}^{B}R^{A}. \tag{5} $$

  When the rotation is represented as a rotation matrix, this is literally the matrix inverse, and since rotation matrices are orthonormal, we also have $R^{-1} = R^{T}$.
- Transforms bundle this up into a single, convenient notation when positions are measured from a frame (and the same frame they are expressed in):

$$ {}^{G}p^{A} = {}^{G}X^{F}\,{}^{F}p^{A} = {}^{G}p^{F} + {}^{F}p^{A}_{G} = {}^{G}p^{F} + {}^{G}R^{F}\,{}^{F}p^{A}. \tag{6} $$

- Transforms compose:

$$ {}^{A}X^{B}{}^{B}X^{C} = {}^{A}X^{C}, \tag{7}$$

and have an inverse

$$ \left[{}^{A}X^{B}\right]^{-1} = {}^{B}X^{A}. \tag{8}$$

Please note that for transforms, we generally do *not* have that X^{-1} is X^{T}, though it still has a simple form.

In practice, transforms are implemented using homogenous coordinates, but for now I'm happy to leave that as an implementation detail.

Example 3.1 (From camera frame to world frame)

Imagine that I have a depth camera mounted in a fixed pose in my workspace. Let's call the camera frame C and denote its pose in the world with ${}^{W}X^{C}$.

A depth camera returns points in the camera frame. Therefore, we'll write this position of point P_i with ${}^{C}p^{P_i}$. If we want to convert the point into the world frame, we simply have

$$ p^{P_i} = X^{CC}p^{P_i}. $$

This is a work-horse operation for us. We often aim to merge points from multiple cameras (typically in the world frame), and always need to somehow relate the frames of the camera with the frames of the robot. The inverse transform, ${}^{C}X^{W}$, which projects world coordinates into the camera frame, is often called the camera "extrinsics".

3.3.1 Representations for 3D rotation

In the spatial algebra above, I've written the rules for rotations using an abstract notion of rotation. But in order to implement this algebra in code, we need to decide how we are going to represent those representations with a small number of real values. There are many possible representations of 3D rotations, they are each good for different operations, and unfortunately, there is no one representation to rule them all. (This is one of the many reasons why everything is better in 2D!) Common representations include

- 3×3 rotation matrices,
- Euler angles (e.g. roll-pitch-yaw),
- axis angle (aka "exponential coordinates"), and
- unit quaternions.

In Drake, we provide all of these representations, and make it easy to convert back and forth between them.

A 3×3 rotation matrix is an orthonormal matrix with columns formed by the $x-$, $y-$, and $z-$axes. Specifically, the first column of the transform ${}^{G}R^{F}$ has the $x-$axis of frame F expressed in G in the first column, etc.

Euler angles specify a 3D rotation by a series of rotations around the x, y, and z axes. The order of these rotations matters; and many combinations can be used to describe any 3D rotation. This is why we use `RollPitchYaw` in the code (preferring it over the more general term "Euler angle") and document it carefully. Roll is a rotation around the x-axis, pitch is a rotation around the y-axis, and yaw is a rotation around the z-axis; this is also known as "extrinsic X-Y-Z" Euler angles.

Axis angles describe a 3D rotation by a scalar rotation around an arbitrary vector axis using three numbers: the direction of the vector is the axis, and the magnitude of the vector is the angle. You can think of unit quaternions as a form of axis angles that have been carefully normalized to be unit length and have magical properties. My favorite careful description of quaternions is probably chapter 1 of [5].

Why all of the representations? Shouldn't "roll-pitch-yaw" be enough? Unfortunately, no. The limitation is perhaps most easily seen by looking at the coordinate changes from roll-pitch-yaw to/from a rotation matrix. Any roll-pitch-yaw can be converted to a rotation matrix, but the inverse map has a singularity. In particular, when the pitch angle is $\frac{\pi}{2}$, then roll and yaw are now indistinguishable. This is described very nicely, along with its physical manifestation in "gimbal lock" in this video. Similarly, the direction of the vector in the axis-angle representation in not uniquely defined when the rotation angle is zero. These singularities become problematic when we start taking derivatives of rotations, for instance when we write the equations of motion. It is now well understood [5] that it requires at least four numbers to properly represent the group of 3D rotations; the unit quaternion is the most common four-element representation.

3.4 Forward kinematics

The spatial algebra gets us pretty close to what we need for our pick and place algorithm. But remember that the interface we have with the robot reports measured joint positions, and expects commands in the form of joint positions. So our remaining task is to convert between joint angles and cartesian frames. We'll do this in steps, the first step is to go from joint positions to cartesian frames: this is known as *forward kinematics*.

Throughout this text, we will refer to the joint positions of the robot (also known as "configuration" of the robot) using a vector q. If the configuration of the scene includes objects in the environment as well as the robot, we would use q for the entire configuration vector, and use e.g. q_{robot} for the subset of the vector corresponding to the robot's joint positions. Therefore, the goal of forward kinematics is to produce a map:

$$X^G = f^G_{kin}(q). \tag{9}$$

Moreover, we'd like to have forward kinematics available for any frame we have defined in the scene. Our spatial notation and spatial algebra makes this computation relatively straight-forward.

3.4.1 The kinematic tree

In order to facilitate kinematics and related multibody computations, the `MultibodyPlant` organizes all of the bodies in the world into a tree topology. Every body (except the world body) has a parent, which it is connected to via either a `Joint` or a "floating base".

Example 3.2 (Inspecting the kinematic tree)

Drake provides some visualization support for inspecting the kinematic tree data structure. The kinematic tree for an iiwa is more of a vine than a tree (it's a serial manipulator), but the tree for the dexterous hands are more interesting. I've added our brick to the example, too, so that you can see that a "free" body is just another branch off the world root node.

Every `Joint` and "floating base" has some number of position variables associated with it -- a subset of the configuration vector q -- and knows how to compute the configuration dependent transform across the joint from the child joint frame J_C

to the parent joint frame J_P: ${}^{J_P}X^{J_C}(q)$. Additionally, the kinematic tree defines the (fixed) transforms from the joint frame to the child body frame, ${}^{C}X^{J_C}$, and from the joint frame to the parent frame, ${}^{P}X^{J_P}$. Altogether, we can compute the configuration transform between any one body and its parent,

$$ {}^{P}X^{C}(q) = {}^{P}X^{J_P}{}^{J_P}X^{J_C}(q){}^{J_C}X^{C}. $$

You might be tempted to think that every time you add a joint to the `MultibodyPlant`, you are adding a degree of freedom. But it actually works the other way around. Every time you add a *body* to the plant, you are adding many degrees of freedom. But you can then add joints to *remove* those degrees of freedom; joints are constraints. "Welding" the robot's base to the world frame removes all of the floating degrees of freedom of the base. Adding a rotational joint between a child body and a parent body removes all but one degree of freedom, etc.

3.4.2 Forward kinematics for pick and place

In order to compute the pose of the gripper in the world, X^G, we simply query the parent of the gripper frame in the kinematic tree, and recursively compose the transforms until we get to the world frame.

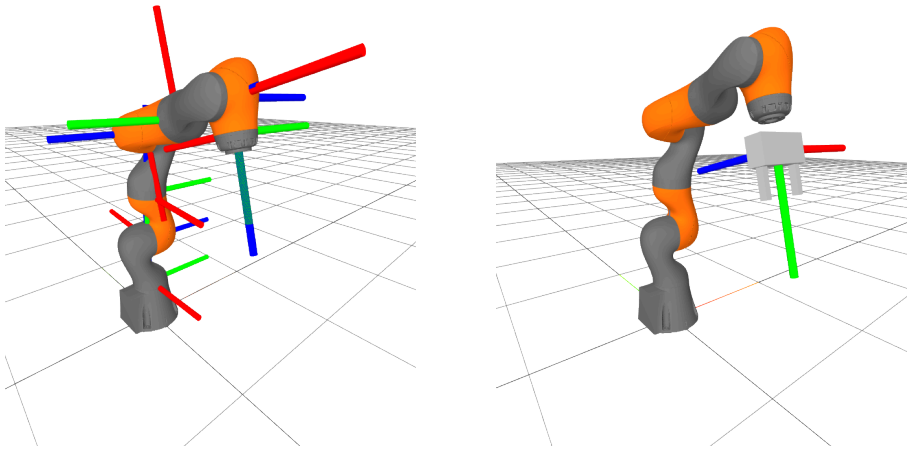


Figure 3.3 - Kinematic frames on the iiwa (left) and the WSG (right). For each frame, the positive x axis is in red, the positive y axis is in green, and the positive z axis is in blue. It's (hopefully) easy to remember: XYZ $\Leftrightarrow$ RGB.

Example 3.3 (Forward kinematics for the gripper frame)

Let's evaluate the pose of the gripper in the world frame: X^G. We know that it will be a function of configuration of the robot, which is just a part of the total state of the `MultibodyPlant` (and so is stored in the `Context`). The following example shows you how it works.

The key lines are

```
gripper = plant.GetBodyByName("body", wsg)
pose = plant.EvalBodyPoseInWorld(context, gripper)
```

Behind the scenes, the `MultibodyPlant` is doing all of the spatial algebra we described above to return the pose (and also some clever caching because you can reuse much of the computation when you want to evaluate the pose of another frame on the same robot).

Example 3.4 (Forward kinematics of "floating-base" objects)

Consider the special case of having a `MultibodyPlant` with exactly one body added, and no joints. The kinematic tree is simply the world frame, the body frame, and they are connected by the "floating base". What does the forward kinematics function:

$$X^B = f^B_{kin}(q),$$

look like in that case? If q is already representing the floating-base configuration, is f^B_{kin} just the identity function?

This gets into the subtle points of how we represent transforms, and how we represent 3D rotations in particular. Although we use rotation matrices in our `RigidTransform` class, in order to make the spatial algebra efficient, we actually use unit quaternions in the `Context` in order to have a more compact vector representation.

As a result, for this example, the software implementation of the function f^B_{kin} is precisely the function that converts the position $\times$ unit quaternion representation into the position $\times$ rotation matrix representation.

3.5 Differential kinematics (Jacobians)

The forward kinematics machinery gives us the ability to compute the pose of the gripper and the pose of the object, both in the world frame. But if our goal is to *move* the gripper to the object, then we should understand how changes in the joint angles relate to changes in the gripper pose. This is traditionally referred to as "differential kinematics".

At first blush, this is straightforward. The change in pose is related to a change in joint positions by the (partial) derivative of the forward kinematics:

$$dX^B = \frac{\partial f^B_{kin}(q)}{\partial q} dq = J^B(q) dq. \tag{10}$$

Partial derivatives of a function are referred to as "Jacobians" in many fields; in robotics it's rare to refer to derivatives of the kinematics as anything else.

All of the subtlety, again, comes in because of the multiple representations that we have for 3D rotations (rotation matrix, unit quaternions, ...). While there is no one best representation for 3D rotations, it *is* possible to have one canonical representation for *differential* rotations. Without any concern for singularities nor loss of generality, we can represent the rate of change in pose using a six-component vector for *spatial velocity*:

$${}^A V^B_C = \begin{bmatrix} {}^A\omega^B_C \\ {}^A\mathrm{v}^B_C \end{bmatrix}. \tag{11}$$

${}^A V^B_C$ is the spatial velocity (also known as a "twist") of frame B measured in frame A expressed in frame C, ${}^A\omega^B_C \in \mathbb{R}^3$ is the *angular velocity* (of frame B measured in A expressed in frame C), and ${}^A\mathrm{v}^B_C \in \mathbb{R}^3$ is the *translational velocity* (along with the same shorthands as for positions). The angular velocity is a 3D vector (with w_x, w_y, w_z components); the magnitude of this vector represents the angular speed and the direction represents the (instantaneous) axis of rotation. It's tempting to think of it as the time derivatives of roll, pitch, and yaw, but that's not true; it can easily be converted into that representation through a nonlinear change of coordinates. Spatial velocities fit nicely into our spatial algebra:

- Angular velocities add (when the frames match):

$$ {}^A\omega^B_F + {}^B\omega^C_F = {}^A\omega^C_F, \tag{12} $$

(this deserves to be verified) and have the additive inverse, ${}^A\omega^C_F = -{}^C\omega^A_F$,.

- Rotations can be used to change between the "expressed-in" frames:

$$ {}^A\mathrm{v}^B_G = {}^GR^F\,{}^A\mathrm{v}^B_F, \qquad {}^A\omega^B_G = {}^GR^F\,{}^A\omega^B_F. \tag{13} $$

- Translational velocities compose across frames with:

$$ {}^A\mathrm{v}^C_F = {}^A\mathrm{v}^B_F + {}^B\mathrm{v}^C_F + {}^A\omega^B_F \times {}^Bp^C_F. \tag{14} $$

This can be derived in a few steps (click the triangle to expand)

Differentiating

$$ {}^Ap^C = {}^AX^B\,{}^Bp^C = {}^Ap^B + {}^AR^B\,{}^Bp^C, $$

yields

$$ \begin{aligned} {}^A\mathrm{v}^C =& {}^A\mathrm{v}^B + {}^A\dot{R}^B\,{}^Bp^C + {}^AR^B\,{}^B\mathrm{v}^C \\ =& {}^A\mathrm{v}^B_A + {}^A\dot{R}^B R^A\,{}^Ap^C_A + {}^B\mathrm{v}^C_A. \end{aligned} \tag{15} $$

Allow me to write $\dot{R}R^{-1}$ for ${}^A\dot{R}^BR^A$ (dropping the frames for a moment). It turns out that $\dot{R}R^{-1}$, is always a skew-symmetric matrix. To see this, differentiate $RR^T = I$ to get

$$ \dot{R}R^T + R\dot{R}^T = 0 \Rightarrow \dot{R}R^T = -R\dot{R}^T \Rightarrow \dot{R}R^T = -(\dot{R}R^T)^T, $$

which is the definition of a skew-symmetric matrix. Any 3×3 skew-symmetric matrix can be parameterized by three numbers (we'll use the three-element vector ω), and can be written as a cross product, so $\dot{R}R^Tp = \omega \times p$.

Multiply the right and left sides by ${}^FR^A$ to change the expressed-in frame, and we have our result.

- This reveals that additive inverse for translational velocities is not obtained by switching the reference and measured-in frames; it is slightly more complicated:

$$ -{}^A\mathrm{v}^B_F = {}^B\mathrm{v}^A_F + {}^A\omega^B_F \times {}^Bp^A_F. \tag{16} $$

.

If you're familiar with "screw theory" (as used in, e.g. [2] and [3]), click the triangle to see how those conventions are related.

Screw theory (as used in, e.g. [2] and [3]) often uses a particular form of our spatial velocity referred to as "spatial velocity in the space frame" / "spatial twists" that can be useful in a number of computations. This quantity is ${}^AV^{B_A}$, where B_A is the frame rigidly attached on body B (so ${}^BV^{B_A} = 0$) that is instantaneously in the same pose as A (so ${}^Ap^{B_A} = 0$, ${}^AR^{B_A} = I$) [3, 3.3.2]. These conditions reduce to

$$ {}^AV^{B_A} = \begin{bmatrix} {}^A\omega^{B_A} \\ {}^A\mathrm{v}^{B_A} \end{bmatrix} = \begin{bmatrix} {}^A\omega^{B_A} + {}^{B_A}\omega^B_A \\ {}^A\mathrm{v}^B + {}^B\mathrm{v}^{B_A}_A + {}^A\omega^B \times {}^Bp^{B_A}_A \end{bmatrix} = \begin{bmatrix} {}^A\omega^B \\ {}^A\mathrm{v}^B - {}^A\omega^B \times {}^Ap^B \end{bmatrix}. \tag{17} $$

There is one more velocity to be aware of: I'll use v to denote the generalized velocity vector of the plant. While a spatial velocity ${}^AV^B$ is six components, a translational or angular velocity, ${}^B\mathrm{v}^C$ or ${}^B\omega^C$, is three components, the generalized velocity vector is whatever size it needs to be to encode the time derivatives of the

configuration variables, q. For the iiwa welded to the world frame, that means it has seven components. I've tried to be careful to typeset each of these v's differently throughout the notes. Almost always the distinction is also clear from the context.

Example 3.5 (Don't assume $\dot{q} \equiv v$)

The unit quaternion representation is four components, but these must form a "unit vector" of length 1. Rotation matrices are 9 components, but they must form an orthonormal matrix with $\det(R) = 1$. It's pretty great that for *changes* in rotation, we can use an *unconstrained* three component vector, what we've called the angular velocity vector, ω. And you really should use it; getting rid of that constraint makes both the math and the numerics better.

But there is one minor nuisance that this causes. We tend to want to think of the generalized velocity as the time derivative of the generalized positions. This works when we have only our iiwa in the model, and it is welded to the world frame. But we cannot assume this in general; not when floating-base rotations are concerned. As evidence, here is a simple example that loads exactly one rigid body into the `MultibodyPlant`, and then prints its `Context`.

The output looks like this:

```
Context
--------
Time: 0
States:
  13 continuous states
    1 0 0 0 0 0 0 0 0 0 0 0 0

plant.num_positions() = 7
plant.num_velocities() = 6
```

You can see that this system has 13 total state variables. 7 of them are positions, q; we use unit quaternions in the position vector. But we have just 6 velocities, v; we use angular velocities in the velocity vector. Clearly, if the length of the vectors don't even match, we do *not* have $\dot{q} = v$.

It's not really any harder; Drake provides the `MultibodyPlant` methods `MapQDotToVelocity` and `MapVelocityToQDot` to go back and forth between them. But you have to remember to use them!

Due to the multiple possible representations of 3D rotation, and the potential difference between $\dot{q}$ and v, there are actually **many** different kinematic Jacobians possible. You may hear the terms "analytic Jacobian", which refers to the explicit partial derivative of the forward kinematics (as written in (10)), and "geometric Jacobian" which replaces 3D rotations on the left-hand side with spatial velocities. In Drake's `MultibodyPlant`, we currently offer the geometric Jacobian versions via

- `CalcJacobianAngularVelocity`,
- `CalcJacobianTranslationalVelocity`, and
- `CalcJacobianSpatialVelocity`,

with each taking an argument to specify whether you'd like the Jacobian with respect to $\dot{q}$ or v. If you really like the analytical Jacobian, you could get it (much less efficiently) using our support for automatic differentiation.

Example 3.6 (Kinematic Jacobians for pick and place)

Let's repeat the setup from above, but we'll print out the Jacobian of the gripper

frame, measured from the world frame, expressed in the world frame.

3.6 Differential inverse kinematics

There is important structure in Eq (10). Make sure you didn't miss it. The relationship between joint velocities and end-effector velocities is (configuration-dependent) linear:

$$V^G = J^G(q)v. \tag{18}$$

Maybe this gives us what we need to produce changes in gripper frame G? If I have a desired gripper frame velocity V^{G_d}, then how about commanding a joint velocity $v = \left[J^G(q)\right]^{-1} V^{G_d}$?

3.6.1 The Jacobian pseudo-inverse

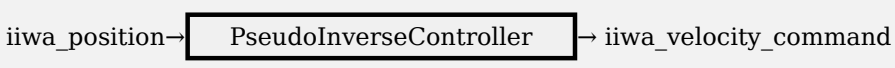
Any time you write a matrix inverse, it's important to check that the matrix is actually invertible. As a first sanity check: what are the dimensions of $J^G(q)$? We know the spatial velocity has six components. Our gripper frame is welded directly on the last link of the iiwa, and the iiwa has seven positions, so we have $J^G(q_{iiwa}) \in \mathbb{R}^{6\times 7}$. The matrix is not square, so does not have an inverse. But having *more* degrees of freedom than the desired spatial velocity requires (more columns than rows) is actually the good case, in the sense that we might have many solutions for v that can achieve a desired spatial velocity. To choose one of them (the minimum-norm solution), we can consider using the Moore-Penrose pseudo-inverse, J^+, instead:

$$v = [J^G(q)]^+ V^{G_d}. \tag{19}$$

The pseudo-inverse is a beautiful mathematical concept. When the J is square and full-rank, the pseudo-inverse returns the true inverse of the system. When there are many solutions (here many joint velocities that accomplish the same end-effector spatial velocity), then it returns the minimum-norm solution (the joint velocities that produce the desired spatial velocity which are closest to zero in the least-squares sense). When there is no exact solution, it returns the joint velocities that produce an spatial velocity which is as close to the desired end-effector velocity as possible, again in the least-squares sense. So good!

Example 3.7 (Our first end-effector "controller")

Let's write a simple controller using the pseudo-inverse of the Jacobian. First, we'll write a new `LeafSystem` that defines one input port (for the iiwa measured position), and one output port (for the iiwa joint velocity command). Inside that system, we'll ask MultibodyPlant for the gripper Jacobian, and compute the joint velocities that will implement a desired gripper spatial velocity.



To keep things simple for this first example, we'll just command a constant gripper spatial velocity, and only run the simulation for a few seconds.

Note that we do have to add one additional system into the diagram. The output of our controller is a desired joint velocity, but the input that the iiwa controller is expecting is a desired joint position. So we will insert an integrator in between.

I don't expect you to understand every line in this example, but it's worth finding the important lines and making sure you can change them and see what happens!

Congratulations! Things are really moving now.

3.6.2 Invertibility of the Jacobian

There is a simple check to understand when the pseudo-inverse can give an exact solution for any spatial velocity (achieving exactly the desired spatial velocity): the Jacobian must be full *row*-rank. In this case, we need $\text{rank}(J) = 6$. But assigning an integer rank to a matrix doesn't tell the entire story; for a real robot with (noisy) floating-point joint positions, as the matrix gets *close* to losing rank, the (pseudo-)inverse starts to "blow-up". A better metric, then, is to watch the smallest singular value; as this approaches zero, the norm of the pseudo-inverse will approach infinity.

Example 3.8 (Invertibility of the gripper Jacobian)

You might have noticed that I printed out the smallest singular value of J^G in one of the previous examples. Take it for another spin. See if you can find configurations where the smallest singular value gets close to zero.

Here's a hint: try some configurations where the arm is very straight, (e.g. driving joint 2 and 4 close to zero).

Another good way to find the singularities are to use your pseudo-inverse controller to send gripper spatial velocity commands that drive the gripper to the limits of the robot's workspace. Try it and see! In fact, this is the common case, and one that we will work hard to avoid.

Configurations q for which $\text{rank}(J(q_{iiwa})) < 6$ for a frame of interest (like our gripper frame G) are called *kinematic singularities*. Try to avoid going near them if you can! The iiwa has many virtues, but admittedly its kinematic workspace is not one of them. Trust me, if you try to get a big Kuka to reach into a little kitchen sink all day, every day, then you will spend a non-trivial amount of time thinking about avoiding singularities.

In practice, things can get a lot better if we stop bolting our robot base to a fixed location in the world. Mobile bases add complexity, but they are wonderful for improving the kinematic workspace of a robot.

Example 3.9 (Are kinematic singularities real?)

A natural question when discussing singularities is whether they are somehow real, or whether they are somehow an artifact of the analysis. Perhaps it is useful to look at an extremely simple case.

Imagine a two-link arm, where each link has length one. Then the kinematics reduce to

$$p^G = \begin{bmatrix} c_0 + c_{0+1} \\ s_0 + s_{0+1} \end{bmatrix},$$

where I've used the (very common) shorthard s_0 for $\sin(q_0)$ and s_{0+1} for $\sin(q_0 + q_1)$, etc. The *translational* Jacobian is

$$J^G(q) = \begin{bmatrix} -s_0 - s_{0+1} & -s_{0+1} \\ c_0 + c_{0+1} & c_{0+1} \end{bmatrix},$$

and as expected, it loses rank when the arm is at full extension (e.g. when $q_0 = q_1 = 0$ which implies the first row is zero).

Click here for the animation.

Let's move the robot along the x-axis, by taking $q_0(t) = 1 - t$, and $q_1(t) = -2 + 2t$. This clearly visits the singularity $q_0 = q_1 = 0$ at time 1, and then leaves again without trouble. In fact, it does all this with a *constant* joint velocity ($\dot{q}_0 = -1, \dot{q}_1 = 2$)! The resulting trajectory is

$$p^G(t) = \begin{bmatrix} 2\cos(1-t) \\ 0 \end{bmatrix}.$$

There are a few things to understand. At the singularity, there is nothing that the robot can do to move its gripper farther in positive x -- that singularity is real. But it is also true that there is no way for the robot to move instantaneously back in the direction of $-x$. The Jacobian analysis is not an approximation, it is a perfect description of the relationship between joint velocities and gripper velocities. However, just because you cannot achieve an instantaneous velocity in the backwards direction, it does not mean you cannot get there! At $t = 1$, even though the joint velocities are constant, and the translational Jacobian is singular, the robot is *accelerating* in $-x$, $\ddot{p}^G_{W_x}(t) = -2\cos(1-t)$.

For the cases when the Jacobian does not have a unique inverse, there is an interesting subtlety to be aware of regarding *integrability*. Let's start our robot in at time zero in a joint configuration, $q(0)$ with a corresponding in an end-effector pose $X^G(0)$. Now let us execute an end-effector trajectory using the pseudo-inverse controller that moves along a closed-path in the task space, coming back to the original end-effector pose at time one: $X^G(1) = X^G(0)$. Unfortunately, there is no guarantee that $q(1) = q(0)$; this is not simply due to numerical errors in our numerical implementation, it is well known that the pseudo-inverse does not in general produce an integrable function. [6] discusses this thoroughly, and proposed a weighted pseudo-inverse that can restore the integrability conditions by means of a virtual "compliance function".

3.7 Defining the grasp and pre-grasp poses

I'm going to put my red foam brick on the table. Its geometry is defined as a 7.5cm x 5cm x 5cm box. For reference, the distance between the fingers on our gripper in the default "open" position is 10.7cm. The "palm" of the gripper is 3.625cm from the body origin, and the fingers are 8.2cm long.

To make things easy to start, I'll promise to set the object down on the table with the object frame's z-axis pointing up (aligned with the world z axis), and you can assume it is resting on the table safely within the workspace of the robot. But I reserve the right to give the object arbitrary initial yaw. Don't worry, you might have noticed that the seventh joint of the iiwa will let you rotate your gripper around quite nicely (well beyond what my human wrist can do).

Observe that visually the box has rotational symmetry -- I could always rotate the box 90 degrees around its x-axis and you wouldn't be able to tell the difference. We'll think about the consequences of that more in the next chapter when we start using perception. But for now, we are ok using the omniscient "cheat port" from the simulator which gives us the unambiguous pose of the brick.

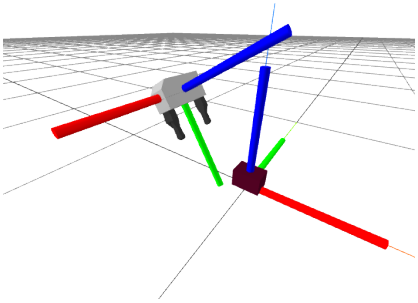


Figure 3.5 - The gripper frame and the object frame. For each frame, the positive x axis is in red, the positive y axis is in green, and the positive z axis is in blue (XYZ $\Leftrightarrow$ RGB).

Take a careful look at the gripper frame in the figure above, using the colors to understand the axes. Here is my thinking: Given the size of the hand and the object, I want the *desired* position (in meters) of the object in the gripper frame,

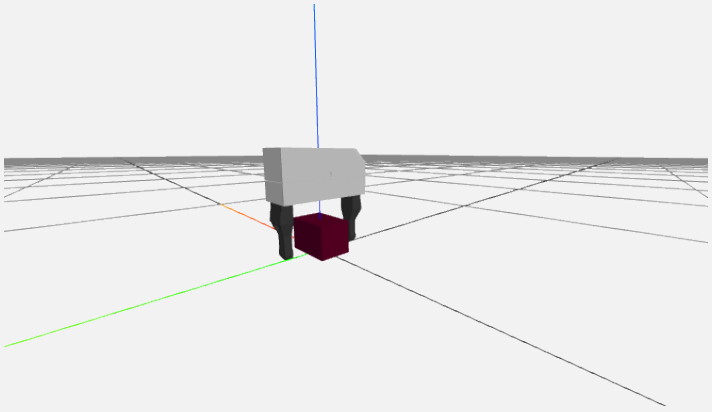
$$ {}^{G_{grasp}}p^{O} = \begin{bmatrix} 0 \\ 0.11 \\ 0 \end{bmatrix}, \qquad {}^{G_{pregrasp}}p^{O} = \begin{bmatrix} 0 \\ 0.2 \\ 0 \end{bmatrix}. $$

Recall that the logic behind a *pregrasp* pose is to first move to safely above the object, if our only gripper motion that is very close to the object is a straight translation from the pregrasp pose to the grasp pose and back, then it allows us to mostly avoid having to think about collisions (for now). I want the orientation of my gripper to be set so that the positive z axis of the object aligns with the negative y axis of the gripper frame, and the positive x axis of the object aligns with the positive z axis of the gripper. We can accomplish that with

$$ {}^{G_{grasp}}R^{O} = \text{MakeXRotation}\left(\frac{\pi}{2}\right)\text{MakeZRotation}\left(\frac{\pi}{2}\right). $$

I admit I had my right hand in the air for that one! Our pregrasp pose will have the same orientation as our grasp pose.

Example 3.10 (Computing grasp and pregrasp poses)



Here is a simple example of loading a floating Schunk gripper and a brick, computing the grasp / pregrasp pose (drawing out each transformation clearly), and rendering the hand relative to the object.

I hope you can see the value of having good notation at work here. My right hand was in the air when I was deciding what a suitable relative pose for the object in the hand should be (when writing the notes). But once that was decided, I went to type it in and everything just worked.

3.8 A PICK AND PLACE TRAJECTORY

We're getting close. We know how to produce desired gripper poses, and we know how to change the gripper pose instantaneously using spatial velocity commands. Now we need to specify how we want the gripper poses to change over time, so that we can convert our gripper poses into spatial velocity commands.

Let's define all of the "keyframe" poses that we'd like the gripper to travel through, and time that it should visit each one. The following example does precisely that.

Example 3.11 (A plan "sketch")

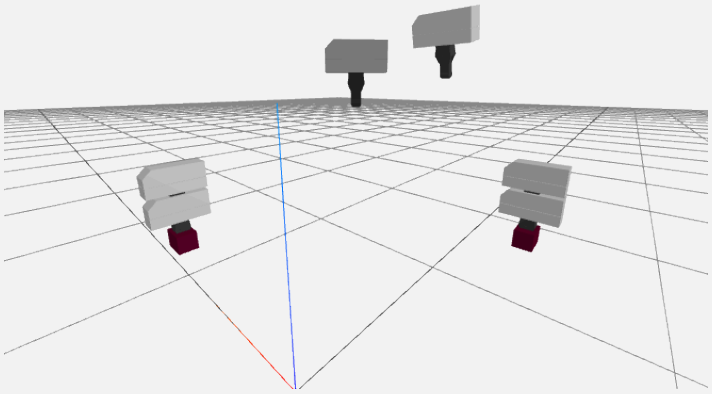


Figure 3.7 - Keyframes of the gripper. The robot's base will be at the origin, so we're looking over the (invisible) robot's shoulder here. The hand starts in the "initial" pose near the center, moves to the "prepick" to "pick" to "prepick" to "clearance" to "preplace" to "place" and finally back to "preplace".

How did I choose the times? I started everything at time, $t = 0$, and listed the rest of our times as absolute (time from zero). That's when the robot wakes up and sees the brick. How long should we take to transition from the starting pose to the pregrasp pose? A really good answer might depend on the exact joint speed limits of the robot, but we're not trying to move fast yet. Instead I chose a conservative time that is proportional to the total Euclidean distance that the hand will travel, say $k = 10\ s/m$ (aka 10 cm/s):

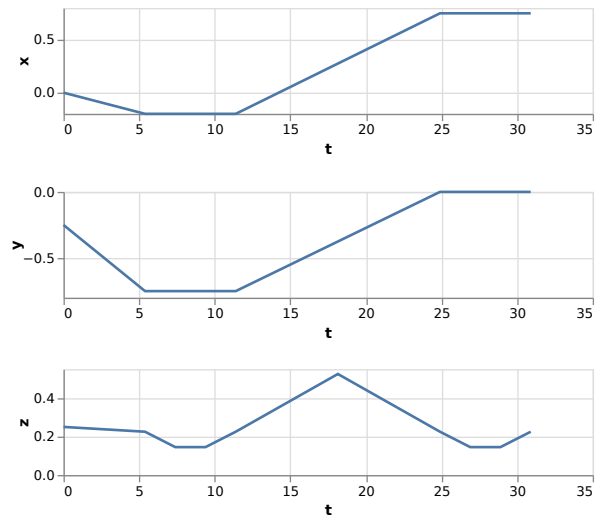
$$t_{pregrasp} = k \left\| {}^{G_0}p^{G_{pregrasp}} \right\|_2.$$

I just chose a fixed duration of 2 seconds for the transitions from pregrasp to grasp and back, and also left 2 seconds with the gripper stationary for the segments where the fingers needs to open/close.

There are a number of ways one might represent a trajectory computationally. We have a pretty good collection of trajectory classes available in Drake. Many of them are implemented as splines -- piecewise polynomial functions of time. Interpolating between orientations requires some care, but for positions we can do a simple linear interpolation between each of our keyframes. That would be called a "first-order hold", and it's implemented in Drake's PiecewisePolynomial class. For rotations, we'll use something called "spherical linear interpolation" or slerp, which is implemented in Drake's PiecewiseQuaternionSlerp, and which you can explore in this exercise. The PiecewisePose class makes it convenient to construct and work with the position and orientation trajectories together.

Example 3.12 (Grasping with trajectories)

There are a number of ways to visualize the trajectory when it's connected to 3D. I've plotted the *position* trajectory as a function of time below.



With 3D data, you can plot it in 3D. But my favorite approach is as an animation in our 3D renderer! Make sure you try the "Open controls>Animation" interface. You can pause it and then scrub through the trajectory using the time slider.

For a super interesting discussion on how we might visualize the 4D quaternions as creatures trapped in 3D, you might enjoy this series of "explorable" videos.

One final detail -- we also need a trajectory of gripper commands (to open and close the gripper). We'll use a first-order hold for that, as well.

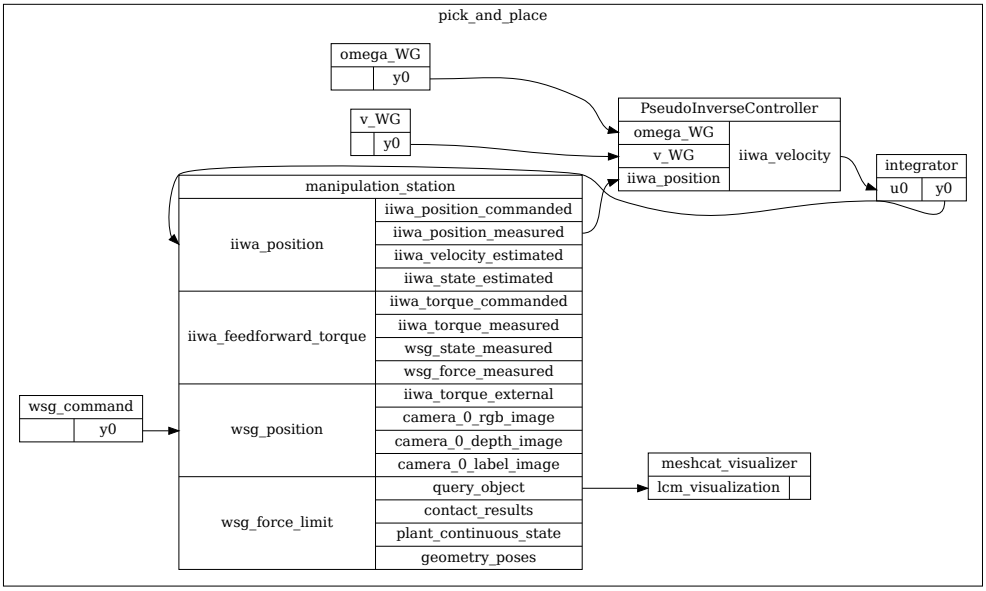
3.9 Putting it all together

We can slightly generalize our PseudoInverseController to have additional input ports for the desired gripper spatial velocity, ${}^{W}V^{G}$ (in our first version, this was just hard-coded in the controller).

The trajectory we have constructed is a *pose* trajectory, but our controller needs spatial velocity commands. Fortunately, the trajectory classes we have used support differentiating the trajectories. In fact, the `PiecewiseQuaternionSlerp` is clever enough to return the derivative of the 4-component quaternion trajectory as a 3-component angular velocity trajectory, and taking the derivative of a `PiecewisePose` trajectory returns a spatial velocity trajectory. The rest is just a matter of wiring up the system diagram.

Example 3.13 (The full pick and place demo)

The next few cells of the notebook should get you a pretty satisfying result. Click here to watch it without doing the work.



It's worth scrutinizing the result. For instance, if you examine the context at the final time, how close did it come to the desired final gripper position? Are you happy with the joint positions? If you ran that same trajectory in reverse, then back and forth (as an industrial robot might), would you expect errors to accumulate?

3.10 DIFFERENTIAL INVERSE KINEMATICS WITH CONSTRAINTS

Our solution above works in many cases. We could potentially move on. But with just a little more work, we can get a much more robust solution... one that we will be happy with for many chapters to come.

So what's wrong with the pseudo-inverse controller? You won't be surprised when I say that it does not perform well around singularities. When the minimum singular value of the Jacobian gets small, that means that some values in the inverse get very large. If you ask our controller to track a seemingly reasonable end-effector spatial velocity, then you might have extremely large velocity commands that result.

There are other important limitations, though, which are perhaps more subtle. The real robot has constraints, very real constraints on the joint angles, velocities, accelerations, and torques. If you, for instance, send a velocity command to the iiwa controller that cannot be followed, that velocity will be clipped. In the mode we are running the iiwa (joint-impedance mode), the iiwa doesn't know anything about your end-effector goals. So it will very likely simply saturate your velocity commands independently joint by joint. The result, I'm afraid, will not be as convenient as a slower end-effector trajectory. Your end-effector could run wildly off course.

Since we know the limits of the iiwa, a better approach is to take these constraints into account at the level of our controller. It's relatively straight-forward to take position, velocity, and acceleration constraints into account; torques would require a full dynamics model so we won't worry about them yet here.

3.10.1 Pseudo-inverse as an optimization

I introduced the pseudo-inverse as having almost magical properties: it returns an exact solution when one is available, or the best possible solution (in the least-

squares norm) when one is not. These properties can all be understood by realizing that the pseudo-inverse is just the optimal solution to a least-squares optimization problem:

$$\min_v \ J^G(q)v - V^{G_d}{}_2^2. \tag{20}$$

When I write an optimization of this form, I will refer to v as the decision variable(s), and I will use v^* to denote the optimal solution (the value of the decision variables that minimizes the cost). Here we have

$$v^* = \left[J^G(q)\right]^+ V^{G_d}.$$

Optimization is an incredibly rich topic, and we will put many tools from optimization theory to use over the course of this text. For a beautiful rigorous but accessible treatment of convex optimization, I highly recommend [7]; it is free online and even reading the first chapter can be incredibly valuable. For a very short introduction to using optimization in Drake, please take a look at the tutorials on "Solving Mathematical Programs" linked from the Drake front page. I use the term "mathematical program" almost synonymously with "optimization problem". Mathematical program is slightly more appropriate if we don't actually have an objective; only constraints.

3.10.2 Adding velocity constraints

Once we understand our existing solution through the lens of optimization, we have a natural route to generalizing our approach to explicitly reason about the constraints. The velocity constraints are the most straight-forward to add.

$$\begin{aligned} \min_v \quad & J^G(q)v - V^{G_d}{}_2^2, \\ \text{subject to} \quad & v_{min} \leq v \leq v_{max}. \end{aligned} \tag{21}$$

You can read this as "find me the joint velocities that achieve my desired gripper spatial velocity as closely as possible, but satisfy my joint velocity constraints." The solution to this can be much better than what you would get from solving the unconstrained optimization and then simply trimming any velocities to respect the constraints after the fact.

This is, admittedly, a harder problem to solve in general. The solution cannot be described using only the pseudo-inverse of the Jacobian. Rather, we are going to solve this (small) optimization problem directly in our controller every time it is evaluated. This problem has a convex quadratic objective and linear constraints, so it falls into the class of convex Quadratic Programming (QP). This is a particularly nice class of optimization problems where we have very strong numerical tools.

Example 3.14 (Jacobian-based control with velocity constraints)

To help think about the differential inverse kinematics problem as an optimization, I've put together a visualization of the optimization landscape of the quadratic program. In the notebook below, you will see the constraints visualized as in red, and the objective function visualized as a function in green.

3.10.3 Adding position and acceleration constraints

We can easily add more constraints to our QP, without significantly increasing the complexity, as long as they are linear in the decision variables. So how should we add constraints on the joint position and acceleration?

The natural approach is to make a first-order approximation of these constraints. To do that, the controller needs some characteristic time step / timescale to relate its velocity decisions to positions and accelerations. We'll denote that time step as h.

The controller already accepts the current measured joint positions q as an input; let us now also take the current measured joint velocities v as a second input. And we'll use v_n for our decision variable -- the next velocity to command. Using a simple Euler approximation of position and first-order derivative for acceleration gives us the following optimization problem:

$$\begin{aligned} \min_{v_n} \quad & J^G(q)v_n - V^{G_d}{}_2^2, \\ \text{subject to} \quad & v_{min} \leq v_n \leq v_{max}, \\ & q_{min} \leq q + hv_n \leq q_{max}, \\ & \dot{v}_{min} \leq \frac{v_n - v}{h} \leq \dot{v}_{max}. \end{aligned} \tag{22}$$

3.10.4 Joint centering

Our Jacobian is 6×7, so we actually have more degrees of freedom than end-effector goals. This is not only an opportunity, but a responsibility. When the rank of J^G is less than the number of columns / number of degrees of freedom, then we have specified an optimization problem that has an infinite number of solutions; and we've left it up to the solver to choose one. Typically a convex optimization solver does choose something reasonable, like taking the "analytic center" of the constraints, or a minimum-norm solution for an unconstrained problem. But why leave this to chance? It's much better for us to completely specify the problem so that there is a unique global optima.

In the rich history of Jacobian-based control for robotics, there is a very elegant idea of implementing a "secondary" control, guaranteed (in some cases) not to interfere with our primary end-effector spatial velocity controller, by projecting it into the *nullspace* of J^G. So in order to fully specify the problem, we will provide a secondary controller that attempts to control *all* of the joints. We'll do that here with a simple joint-space controller $v = K(q_0 - q)$; this is a proportional controller that drives the robot to its nominal configuration.

Denote $P(q)$ as an orthonormal basis for the kernel of a Jacobian J. Traditionally in robotics we implemented this using the pseudo-inverse: $P = (I - J^+J)$, but many linear algebra packages now provide methods to obtain one more directly. Adding $Pv \approx Pk(q_0 - q)$ as a secondary objective can be accomplished with

$$\begin{aligned} \min_{v_n} \quad & J^G(q)v_n - V^{G_d}{}_2^2 + \epsilon|P(q)\left(v_n - K(q_0 - q)\right)|_2^2, \\ \text{subject to} \quad & \text{constraints}. \end{aligned} \tag{23}$$

Note the scalar ϵ that we've placed in front of the secondary objective. There is something important to understand here. If we do not have any constraints, then we can remove ϵ completely -- the secondary task will in no way interfere with the primary task of tracking the spatial velocity. However, if there are constraints, then these constraints can cause the two objectives to clash (Exercise). So we pick $\epsilon \ll 1$ to give the primary objective relatively more weight. But don't make it too small, because that might make the numerics bad for your solver. I'd say $\epsilon = 0.01$ is just about right.

There are more sophisticated methods if one wishes to establish a strict task prioritization in the presence of constraints (e.g. [8, 9]), but for th simple prioritization we have formulated here, the penalty method is quite reasonable.

3.10.5 Alternative formulations

Once we have embraced the idea of solving a small optimization problem in our

control loop, many other formulations are possible, too. You will find many in the literature. Minimizing the least-squares distance between the commanded spatial velocity and the resulting velocity might not actually be the best solution. The formulation we have been using heavily in Drake adds an additional constraint that our solution **must** move in the same *direction* as the commanded spatial velocity. If we are up against constraints, then we may slow down, but we will not deviate (instantaneously) from the commanded path. It would be a pity to spend a long time carefully planning collision-free paths for your end-effector, only to have your controller treat the path as merely a suggestion. Note, however, that your plan playback system still needs to be smart enough to realize that the slow-down occurred (open-loop velocity commands are not enough).

$$\begin{aligned} \max_{v_n, \alpha} \quad & \alpha, \\ \text{subject to} \quad & J^G(q)v_n = \alpha V^{G_d}, \\ & 0 \leq \alpha \leq 1, \\ & \text{additional constraints.} \end{aligned} \tag{24}$$

You should immediately ask yourself: is it reasonable to scale a spatial velocity by a scalar? That's another great exercise.

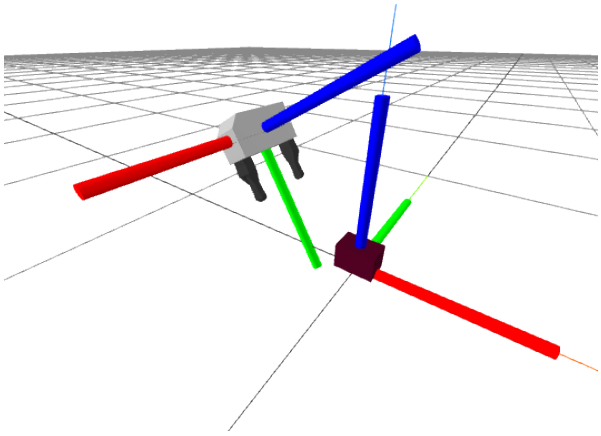
What happens in this formulation when the Jacobian drops row rank? Observe that $v_n = 0, \alpha = 0$ is always a feasible solution for this problem. So if it's not possible to move in the commanded direction, then the robot will just stop.

Example 3.15 (Drake's DifferentialInverseKinematics)

We will use this implementation of differential inverse kinematics whenever we need to command the end-effector in the next few chapters.

3.11 Exercises

Exercise 3.1 (Spatial frames and positions.)



I've rendered the gripper and the brick with their corresponding body frames. Given the configuration displayed in the figure, which is a possible value for ${}^G p^O$?

a. [0.2, 0, -.2]
b. [0, 0.3, .1]
c. [0, -.3, .1]

Which is a possible value for ${}^G p^O_W$?

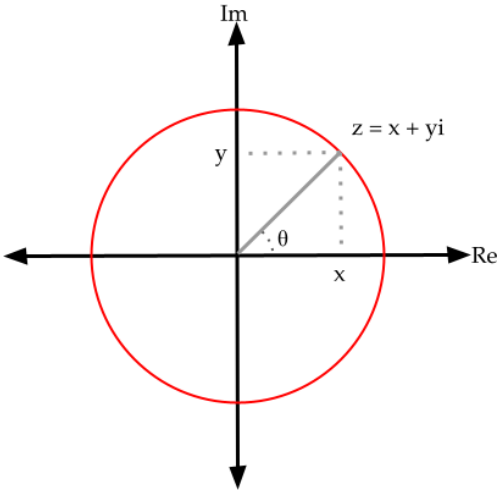
Exercise 3.2 (The additive property of angular velocity vectors.)

TODO(russt): Fill this in.

Exercise 3.3 (Spherical linear interpolation (slerp))

For positions, we can linearly interpolate between points, i.e. a "first-order hold". When dealing with rotations, we cannot simply linearly interpolate and must instead use spherical linear interpolation (slerp). The goal of this problem is to dig into the details of slerp.

To do so we will consider the simpler case where our rotations are in $\mathbb{R}^2$ and can be represented with complex numbers. Here are the rules of the game: a 2D vector (x,y) will be represented as a complex number $z = x + yi$. To rotate this vector by θ, we will multiply by $e^{i\theta} = \cos(\theta) + i\sin(\theta)$.



a. Let's verify that this works. Take the 2D vector $(x, y) = (1, 1)$. If you convert this vector into a complex number, multiply by $z = e^{i\pi/4}$ using complex multiplication, and convert the result back to a 2D vector, do you get the expected result? Show your work and explain why this is the expected result.

Take a minute to convince yourself that this recipe (going from a 2D vector to a complex number, multiply by $e^{i\theta}$, and converting back to a 2D vector) is mathematically equivalent to multiplying the original number by the 2D rotation matrix:

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

A frame F has an orientation. We can represent that orientation using the rotation from the world frame, e.g. ${}^{W}R^{F}$. We've just verified that you can represent that rotation using a complex number, $e^{i\theta}$. Now assume we want to interpolate between two orientations, represented by frames F and G. How should we smoothly interpolate between the two frames, e.g. using $t \in [0, 1]$? You'll explore this in parts (b) and (c).

b. Attempt 1: Consider $z(t) = (a_F * (1 - t) + a_G * t) + i(b_F * (1 - t) + b_G * t)$. Take $t = .5$. What happens if you multiply the 2D vector $(1, 1)$ by $z(t = .5)$? Show your work and explain what goes wrong.

c. Attempt 2: Instead we can leverage the other representation: $z(t) = e^{i\theta_F * (1-t) + i\theta_G * t}$. What happens if we multiply by the 2D vector $(1, 1)$ by $z(t = .5)$? Again, show your work.

Quaternions simply are a generalization of this idea to 3D. In 2D, it might seem inefficient to use two numbers and a constraint that $a^2 + b^2 = 1$ to represent a single rotation (why not just θ!?), but we've seen that it works. In 3D we can use 4 numbers x, y, z, w plus the constraint that $x^2 + y^2 + z^2 + w^2 = 1$ to represent a 3D rotation. In 2D, using just θ can work fine, but in 3D using only three numbers leads to problems like the famous gimbal lock. The 4 numbers forming a unit quaternion provides a non-degenerate mapping of all 3D rotations.

Just like we saw in 2D, one cannot simply linearly interpolate the 4 numbers of the quaternion to interpolate orientations. Instead, we linearly interpolate the angle between two orientations, using the quaternion slerp. The details involve some quaternion notation, but the concept is the same as in 2D.

Exercise 3.4 (Scaling spatial velocity)

TODO(russt): Fill this in. Until then, here's a little code that might convince you it's reasonable.

figures/scaling_spatial_velocity.py

Exercise 3.5 (Planar Manipulator)

For this exercise you will derive the translational forward kinematics ${}^A p^C = f(q)$ and the translational Jacobian $J(q)$ of a planar two-link manipulator. You will work exclusively in . You will be asked to complete the following steps:

a. Derive the forward kinematics of the manipulator.
b. Derive the Jacobian matrix of the manipulator.
c. Analyze the kinematic singularities of the manipulator from the Jacobian.

Exercise 3.6 (Exploring the Jacobian)

Exercise 3.5 asked you to derived the translational Jacobian for the planar two-link manipulator. In this problem we will explore the translational Jacobian in more detail, both in the context of a planar two-link manipulator and in the context of a planar three-link manipulator. For the planar three-link manipulator, the joint angles are (q_0, q_1, q_2) and the planar end-effector position is described by (x, y).

a. For the planar two-link manipulator, the size of the planar translational Jacobian is 2x2. What is the size of the planar translational Jacobian of the planar three-link manipulator?

b. In considering the planar two-link and three-link manipulators, how does the size of the translational Jacobian impact the type of inverse that can be computed? (when can the inverse be computed exactly? when can it not?)

c. Below, for the planar two-link manipulator, we draw the unit circle of joint velocities in the $\dot{\theta}_1 - \dot{\theta}_2$ plane. This circle is then mapped through

the translational Jacobian to an ellipse in the end effector velocity space. In other words, this visualizes that the translational Jacobian maps the space of joint velocities to the space of end-effector velocities.

The ellipse in the end-effector velocity space is called the *manipulability ellipsoid*. The manipulability ellipsoid graphically captures the robot's ability to move its end effector in each direction. For example, the closer the ellipsoid is to a circle, the more easily the end effector can move in arbitrary directions. When the robot is at a singularity, it cannot generate end effector velocities in certain directions. Thinking back to the singularities you explored in Exercise 3.5, at one of these singularities, what shape would the manipulability ellipsoid collapse to?

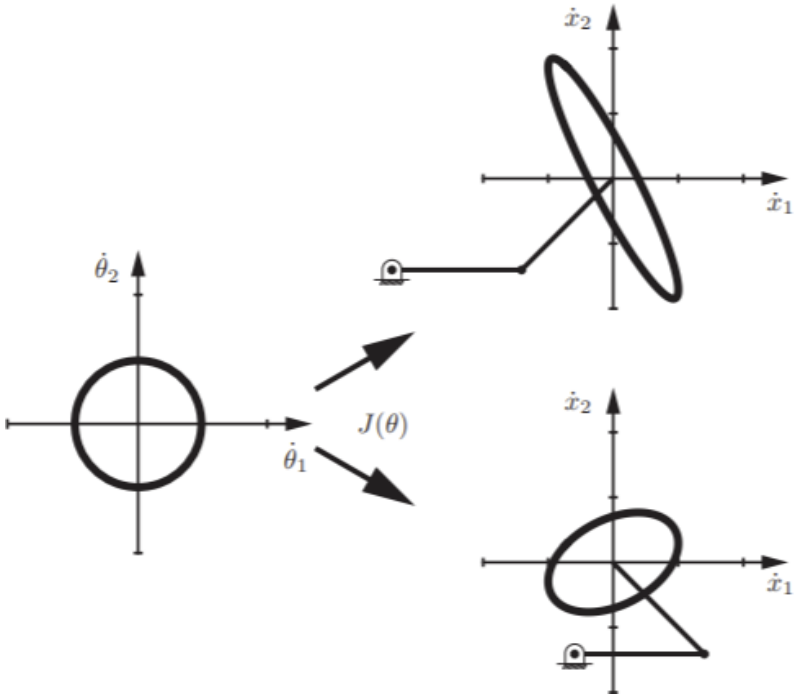


Figure 3.12 - Manipulability ellipsoids for two different postures of the planar two-link manipulator. Source: Lynch, Kevin M., and Frank C. Park. Modern robotics. Cambridge University Press, 2017.

Exercise 3.7 (Spatial Transforms and Grasp Pose)

For this exercise you will apply your knowledge on spatial algebra to write poses of frames in different reference frames, and design a grasp pose yourself. You will work exclusively in . You will be asked to complete the following steps:

a. Express poses of frames in different reference frames using spatial algebra.
b. Design grasp poses given the configuration of the target object and griper configuration.

Exercise 3.8 (The Robot Painter)

For this exercise you will design interesting trajectories for the robot to follow, and observe the robot virtually painting in the air! You will work exclusively in . You will be asked to complete the following steps:

a. Design and compute the poses of key frames of a designated trajectory.
b. Construct trajectories by interpolating through the key frames.

Exercise 3.9 (Introduction to QPs)

For this exercise you will practice the syntax of solving Quadratic Programs (QPs) via Drake's MathematicalProgram interface. You will work exclusively in .

Exercise 3.10 (Virtual Wall)

For this exercise you will implement a virtual wall for a robot manipulator, using an optimization-based approach to differential inverse kinematics. You will work exclusively in . You will be asked to complete the following steps:

a. Implement a optimization-based differential IK controller with joint velocity limits.
b. Using your own constraints, implement a virtual wall in the end-effector space using optimization-based differential IK controller.

Exercise 3.11 (Competing objectives)

In the section on joint centering, I claimed that an secondary objective might compete with a primary objective if they are linked through constraints. To see this, consider the following optimization problem

$$\min_{x,y} \quad (x-5)^2 + (y+3)^2.$$

Clearly, the optimal solution is given by $x^* = 5, y^* = -3$. Moreover, the objective are separable. The addition of the second objective, $(y+3)^2$, did not in any way impact the solution of the first.

Now consider the constrained optimization

$$\begin{aligned} \min_{x,y} \quad & (x-5)^2 + (y+3)^2 \\ \text{subject to} \quad & x - y \leq 6. \end{aligned}$$

If the second objective was removed, then we would still have $x^* = 5$. What is the result of the optimization as written (it's only a few lines of code, if you want to do it that way)? I think you'll find that these "orthogonal" objectives actually compete!

What happens if you change the problem to

$$\begin{aligned} \min_{x,y} \quad & (x-5)^2 + \frac{1}{100}(y+3)^2 \\ \text{subject to} \quad & x - y \leq 6? \end{aligned}$$

I think you'll find that solution is quite close to $x^* = 5$, but also that y^* is quite different than that $y^* = 0$ one would obtain if the "secondary" objective was omitted completely.

Note that if the constraint was not active at the optimal solution (e.g. $x = 5, y = -3$ satisfied the constraints), then the objectives do not compete.

This seems like an overly simple example. But I think you'll find that it is actually

quite similar to what is happening in the nullspace objective formulation above.

Exercise 3.12 (Spatial Velocity for Moving Frame)

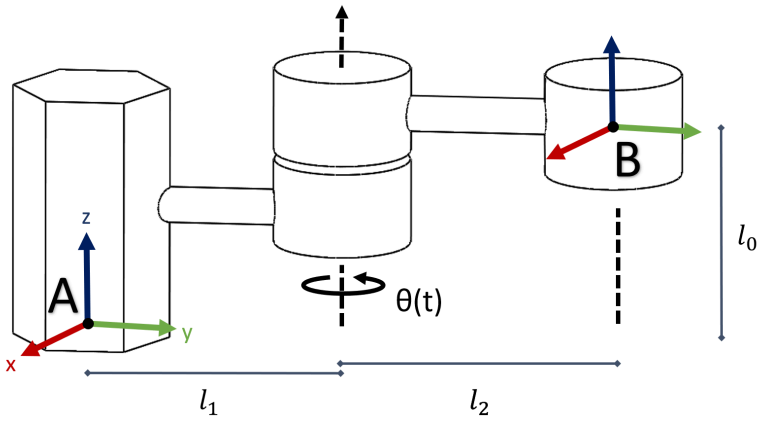


Figure 3.13 - A manipulator with one moving joint. Frame A is the robot base while frame B is on the end-effector.

A manipulator with one joint is shown above. The joint is rotating over time, with its angle as a function of time, $\theta(t)$. When $t = 0, \theta(t) = 0$ and the link(arm) is aligned with y-axis of frame A and that of frame B. For this exercise, we will explore the spatial velocity of the end-effector.

a. For this manipulator, given ${}^{B}p^{C}(0) = [1, 0, 0]^{T}$, write ${}^{A}p^{C}(0)$. For a generic point C, derive 3x3 rotation matrix ${}^{A}R^{B}(t)$ and translation vector ${}^{A}p^{B}(t)$, such that ${}^{A}p^{C}_{A}(t) = {}^{A}R^{B}(t){}^{B}p^{C}_{B} + {}^{A}p^{B}(t)$. Your solutions should involve l_0, l_1, l_2, and $\theta(t)$.
b. Prove that for any rotation matrix R, $\widehat{\omega} \equiv \dot{R}R^{-1}$ satisfies $\widehat{\omega} = -\widehat{\omega}^{T}$. Plug in $R = {}^{A}R^{B}(t)$ to verify your proof.
c. Solve for ${}^{A}V^{B}(t)$ for the manipulator, as a function of $l_0, l_1, l_2, \theta(t)$ and $\dot{\theta}(t)$.

References

1. John J. Craig, "Introduction to Robotics: Mechanics and Control", Pearson Education, Inc , 2005.

2. Richard M. Murray and Zexiang Li and S. Shankar Sastry, "A Mathematical Introduction to Robotic Manipulation", CRC Press, Inc. , 1994.

3. Kevin M Lynch and Frank C Park, "Modern Robotics", Cambridge University Press , 2017.

4. Paul Mitiguy, "Advanced {Dynamics} \& {Motion} {Simulation}: {For} {Professional} {Engineers} and {Scientists} (graduate {Work}) ; {3D}, {Computational}, {Guided}", Prodigy Press , 2017.

5. John Stillwell, "Naive {L}ie theory", Springer Science \& Business Media , 2008.

6. Ferdinando A. Mussa-Ivaldi and Neville Hogan, "Integrable Solutions of Kinematic Redundancy via Impedance Control", *The International Journal of Robotics Research*, vol. 10, no. 5, pp. 481-491, 1991.

7. Stephen Boyd and Lieven Vandenberghe, "Convex Optimization", Cambridge University Press , 2004.

8. Fabrizio Flacco and Alessandro De Luca and Oussama Khatib, "Control of redundant robots under hard joint constraints: Saturation in the null space", *IEEE Transactions on Robotics*, vol. 31, no. 3, pp. 637--654, 2015.

9. Adrien Escande and Nicolas Mansard and Pierre-Brice Wieber, "Hierarchical quadratic programming: Fast online humanoid-robot motion generation", *The International Journal of Robotics Research*, vol. 33, no. 7, pp. 1006--1028, 2014.

Chapter 4
Geometric Pose Estimation

In the last chapter, we developed an initial solution to moving objects around, but we made one major assumption that would prevent us from using it on a real robot: we assumed that we knew the initial pose of the object. This chapter is going to be our first pass at removing that assumption, by developing tools to estimate that pose using the information obtained from the robot's depth cameras. The tools we develop here will be most useful when you are trying to manipulate *known objects* (e.g. you have a mesh file of their geometry) and are in a relatively *uncluttered* environment. But they will also form a baseline for the more sophisticated methods we will study.

The approach that we'll take here is very geometric. This is in contrast to, and very complimentary with, approaches that are more fundamentally driven by data. There is no question that deep learning has had an enormous impact in perception -- it's fair to say that it has enabled the current surge in manipulation advances -- and we will certainly cover it in these notes. But when I've heard colleagues say that "all perception these days is based on deep learning", I can't help but cry foul. There has been another surge of progress in the last few years that has been happening in parallel: the revolution in geometric reasoning, fueled by applications in autonomous driving and virtual/augmented reality in addition to manipulation. Most advanced manipulation systems these days combine both "deep perception" and "geometric perception".

4.1 Cameras and depth sensors

Just as we had many choices when selecting the robot arm and hand, we have many choices for instrumenting our robot/environment with sensors. Even more so than our robot arms, the last few years have seen incredible improvements in the quality and reductions in cost and size for these sensors. This is largely thanks to the cell phone industry, but the race for autonomous cars has been fueling high-end sensors as well.

These changes in hardware quality have caused sometimes dramatic changes in our algorithmic approaches. For example, estimation can be much easier when the resolution and frame rate of our sensors is high enough that not much can change in the world between two images; this undoubtedly contributed to the revolutions in the field of "simultaneous localization and mapping" (SLAM) we have seen over the last decade or so.

One might think that the most important sensors for manipulation are the touch sensors (you might even be right!). But in practice today, most of the emphasis is on camera-based and/or range sensing. At very least, we should consider this first, since our touch sensors won't do us much good if we don't know where in the world we need to touch.

Traditional cameras, which we think of as a sensor that outputs a color image at some framerate, play an important role. But robotics makes heavy use of sensors that make an explicit measurement of the distance (between the camera and the world) or depth; sometimes in addition to color and sometimes in lieu of color. Admittedly, some researchers think we should only rely on color images.

4.1.1 Depth sensors

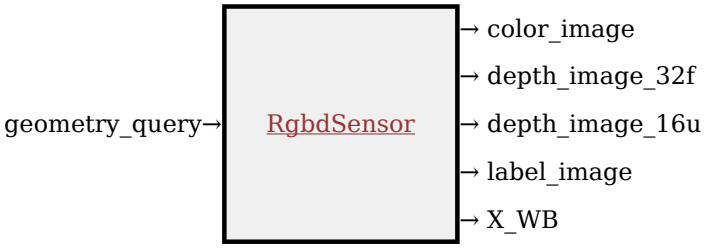
Primarily RGB-D (ToF vs projected texture stereo vs ...) cameras and Lidar

The cameras we are using in this course are Intel RealSense D415.

Monocular depth.

4.1.2 Simulation

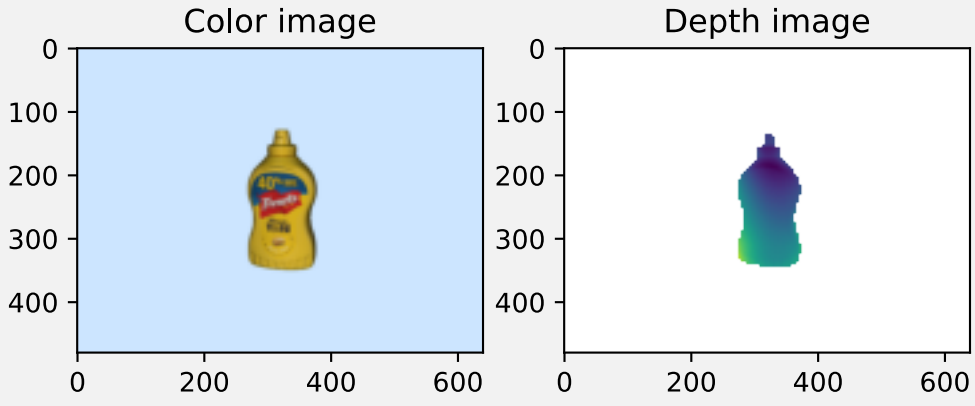
There are a number of levels of fidelity at which one can simulate a camera like the D415. We'll start our discussion here using an "ideal" RGB-D camera simulation -- the pixels returned in the depth image represent the true geometric depth in the direction of each pixel coordinate. In DRAKE that is represented by the `RgbdSensor` system, which can be wired up directly to the `SceneGraph`.



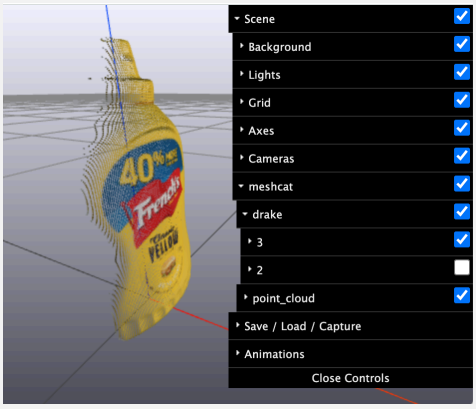
The signals and systems abstraction here is encapsulating a lot of complexity. Inside the implementation of that system is a complete rendering engine, like one that you will find in high-end computer games. Drake actually supports multiple rendering engines; for the purposes of this class we will primarily use an OpenGL-based renderer that is suitable for real-time simulation. In our research we also use Drake's rendering API to connect to high-end physically-based rendering (PBR) based on ray tracing, such as the Cycles renderer provided by Blender. These are most useful when we are trying to render higher quality images e.g. to *train* a deep perception system; we will discuss them in the deep perception chapters.

Example 4.1 (Simulating an RGB-D camera)

As a simple example of depth cameras in drake, I've constructed a scene with a single object (the mustard bottle from the YCB dataset), and added an `RgbdSensor` to the diagram. Once this is wired up, we can simply evaluate the output ports in order to obtain the color and depth images:



Please make sure you spend a minute with the MeshCat visualization (available here). You'll see a camera and the camera frame, and you'll see the mustard bottle as always... but it might look a little funny. That's because I'm displaying both the ground truth mustard bottle model *and* the point cloud rendered from the cameras. You can use the MeshCat GUI to uncheck the ground truth visualization (image right), and you'll be able to see just the point cloud.



Remember that, in addition to looking at the source code if you like, you can always inspect the block diagram to understand what is happening at the "systems level". Here is the diagram used in this example.

In the `ManipulationStation` simulation of the entire robot system for class, the `RgbdSensors` have already been added, and their output ports exposed as outputs for the station.

4.1.3 Sensor noise and depth dropouts

Real depth sensors, of course, are far from ideal -- and errors in depth returns are not simple Gaussian noise, but rather are dependent on the lighting conditions, the surface normals of the object, and the visual material properties of the object, among other things. The color channels are an approximation, too. Even our high-end renderers can only do as well as the geometries, materials and lighting sources that we add to our scene, and it is very hard to capture all of the nuances of a real environment. We will examine real sensor data, and the gaps between modeled sensors and real sensors, after we start understanding some of the basic operations.

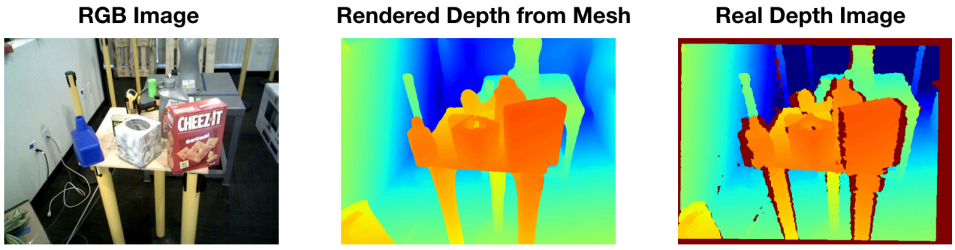


Figure 4.2 - Real depth sensors are messy. The red pixels in the far right image are missed returns -- the depth sensor returned "maximum depth". Missed returns, especially on the edges of objects (or on reflective objects) are a common sighting in raw depth data. This figure is reproduced from [1].

4.1.4 Occlusions and partial views

The view of the mustard bottle in the example above makes one primary challenge of working with cameras very clear. Cameras don't get to see everything! They only get line of sight. If you want to get points from the back of the mustard bottle, you'll need to move the camera. And if the mustard bottle is sitting on the table, you'll have to pick it up if you ever want to see the bottom. This situation gets even worse in cluttered scenes, where we have our views of the bottle *occluded* by other objects. Even when the scene is not cluttered, a head-mounted or table-mounted camera is often blocked by the robot's hand when it goes to do manipulation -- the moments when you would like the sensors to be at their best is often when they give no information!

It is quite common to mount a depth camera on the robot's wrist in order to help with the partial views. But it doesn't completely resolve the problem. All of our depth sensors have both a maximum range and a minimum range. The RealSense D415 returns matches from about 0.3m to 1.5m. That means for the last 30cm of the approach to the object -- again, where we might like our cameras to be at their best -- the object effectively disappears!

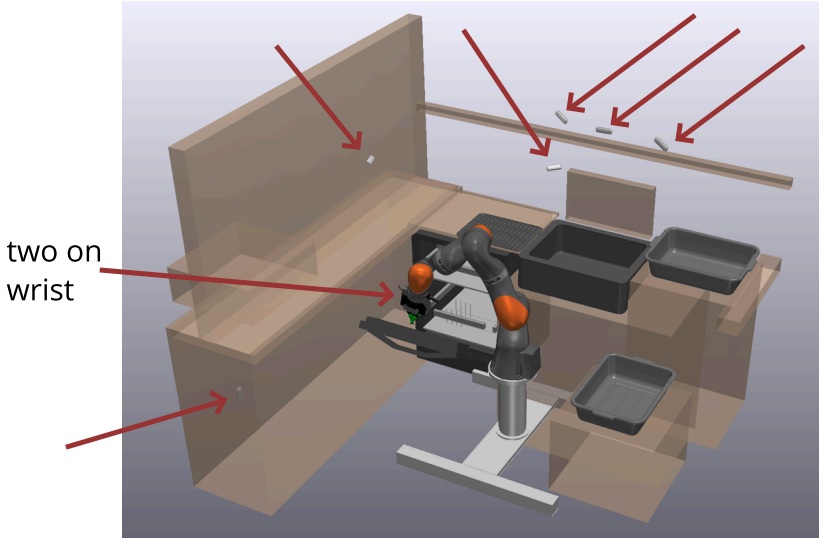


Figure 4.3 - When luxury permits, we try to put as many cameras as we can into the robot's environment. In this dish-loading testbed at TRI where we put a total of eight cameras into the scene (it still doesn't resolve the occlusion problem).

4.2 Representations for Geometry

There are many different representations for 3D geometry, each of which can be more or less suitable for different computations. Often we can convert between these representations; though sometimes that conversion can be lossy. Examples include triangulated surface meshes and tetrahedral volumetric meshes, voxel/occupancy grids, and implicit representations of geometry like the signed distance fields, which have become very popular again recently as representations in deep learning for 3D vision[2, 3]. But the representations that we will start with here are *depth images* and *point clouds*.

The data returned by a depth camera takes the form of an image, where each pixel value is a single number that represents the distance between the camera and the nearest object in the environment along the pixel direction. If we combine this with the basic information about the camera's intrinsic parameters (e.g. lens parameters, stored in the `CameraInfo` class in Drake) then we can transform this depth image representation into a collection of 3D points, s_i. I use s here because they are commonly referred to as the "scene points" in the algorithms we will present below. These points have a pose and (optionally) a color value or other information attached; this is the *point cloud* representation. By default, their pose is described in the camera frame, ${}^{C}X^{s_i}$, but the `DepthImageToPointCloud` system in Drake also accepts ${}^{P}X^{C}$ to make it easy to transform them into another frame (such as the world frame).

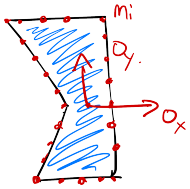
As depth sensors have become so pervasive the field has built up libraries of tools for performing basic geometric operations on point clouds, and that can be used to

transform back and forth between representations. We implement many of of the basic operations directly in Drake. There are also open-source tools like the Open3D library that are available if you need more. Many older projects used the Point Cloud Library (PCL), which is now defunct but still has some very useful documentation.

It's important to realize that the conversion of a depth image into a point cloud does lose some information -- specifically the information about the ray that was cast from the camera to arrive at that point. In addition to declaring "there is geometry at this point", the depth image combined with the camera pose also implies that "there is no geometry in the straight line path between camera and this point". We will make use of this information in some of our algorithms, so don't discard the depth images completely! More generally, we will find that each of the different geometry representations have strengths and weaknesses -- it is very common to keep multiple representations around and to convert back and forth between them.

4.3 Point cloud registration with known correspondences

Let us begin to address the primary objective of the chapter -- we have a known object somewhere in the robot's workspace, we've obtained a point cloud from our depth cameras. How do we estimate the pose of the object, X^O?



One very natural and well-studied formulation of this problem comes from the literature on point cloud registration (also known as point set registration or scan matching). Given two point clouds, point cloud registration aims to find a rigid transform that optimally aligns the two point clouds. For our purposes, that suggests that our "model" for the object should also take the form of a point cloud (at least for now). We'll describe that object with a list of *model points*, m_i, with their pose described in the object frame: ${}^O X^{m_i}$.

Our second point cloud will be the *scene points*, s_i, that we obtain from our depth camera, transformed (via the camera intrinsics) into the camera coordinates, ${}^C X^{s_i}$. I'll use N_m and N_s for the number of model and scene points, respectively.

Let us assume, for now, that we also know X^C. Is this reasonable? In the case of a wrist-mounted camera, this would amount to solving the forward kinematics of the arm. In an environment-mounted camera, this is about knowing the pose of the cameras in the world. Small errors in estimating those poses, though, can lead to large artifacts in the estimated point clouds. We therefore take great care to perform camera extrinsics calibration; I've linked to our calibration code in the appendix. Note that if you have a mobile manipulation platform (an arm mounted on a moving base), then all of these discussions still apply, but you likely will perform all of your registration in the robot's frame instead of the world frame.

The second **major assumption** that we will make in this section is "known correspondences". When I say "correspondence" here, I mean here that for each point in the scene point cloud, we can identify it with a specific point in the model point cloud; for instance, this might be the case if each point had a unique color that we could perceive reliable through the camera. This is *not* a reasonable assumption in practice, but it helps us get started. To represent this mapping in our equations, I'll use a correspondence vector $c \in [1, N_m]^{N_s}$, where $c_i = j$ denotes that scene point s_i corresponds with model point m_j. Note that we do not assume that these correspondences are one-to-one. Every scene point corresponds to some model point, but the converse is not true, and multiple scene points may correspond to the same model point.

As a result, the point cloud registration problem is simply an (inverse) kinematics problem. We can write the model points and the scene points in a common frame (here the world frame),

$$p^{m_{c_i}} = X^O \, {}^O p^{m_{c_i}} = X^C \, {}^C p^{s_i},$$

leaving a single unknown transform, X^O, for which we must solve. In the previous chapter I argued for using differential kinematics instead of inverse kinematics; why is my story different here? Differential kinematics can still be useful for perception, for example if we want to track the pose of an object after we acquire its initial pose. But unlike the robot case, where we can read the joint sensors to get our current state, in perception we need to solve this harder problem at least once.

What does a solution for X^O look like? Each model point gives us three constraints, when $p^{s_i} \in \mathbb{R}^3$. The exact number of unknowns depends on the particular representation we choose for the pose, but almost always this problem is dramatically over-constrained. Treating each point as hard constraints on the relative pose would also be very susceptible to measurement noise. As a result, it will serve us better to try to find a pose that describes the data, e.g., in a least-squares sense:

$$\min_{X \in \text{SE}(3)} \sum_{i=1}^{N_s} \| X \, {}^O p^{m_{c_i}} - X^C \, {}^C p^{s_i} \|^2.$$

Here I have used the notation SE(3) for the "special Euclidean group," which is the standard notation saying the X must be a valid rigid transform.

To proceed, let's pick a particular representation for the pose to work with. I will use 3x3 rotation matrices here; the approach I'll describe below also has an equivalent in quaternion form[4]. To write the optimization above using the coefficients of the rotation matrix and the translation as decision variables, I have:

$$\begin{aligned} \min_{p,R} \quad & \sum_{i=1}^{N_s} \| p + R \, {}^O p^{m_{c_i}} - X^C \, {}^C p^{s_i} \|^2, \\ \text{subject to} \quad & R^T = R^{-1}, \quad \det(R) = +1. \end{aligned} \tag{1}$$

The constraints are needed because not every 3x3 matrix is a valid rotation matrix; satisfying these constraints ensures that R *is* a valid rotation matrix. In fact, the constraint $R^T = R^{-1}$ is almost enough by itself; it implies that $\det(R) = \pm 1$. But a matrix that satisfies that constraint with $\det(R) = -1$ is called an "improper rotation", or a rotation plus a reflection across the axis of rotation.

Let's think about the type of optimization problem we've formulated so far. Given our decision to use rotation matrices, the term $p + R^O p^{m_{c_i}} - X^C {}^C p^{s_i}$ is linear in the decision variables (think $Ax \approx b$), making the objective a convex quadratic. How about the constraints? The first constraint is a quadratic equality constraint (to see it, rewrite it as $RR^T = I$ which gives 9 constraints, each quadratic in the elements of R). The determinant constraint is cubic in the elements of R.

Example 4.2 (Rotation-only point registration in 2D)

I would like you to have a graphical sense for this optimization problem, but it's hard to plot the objective function with 9+3 decision variables. To whittle it down to the essentials that I can plot, let's consider the case where the scene points are known to differ from the model points by a rotation only (this is famously known as the Wahba problem). To further simplify, let's consider the problem in 2D instead of 3D.

In 2D, we can actually linearly parameterize a rotation matrix with just two variables:

$$R = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}.$$

With this parameterization, the constraints $RR^T = I$ and the determinant constraints are identical; they both require that $a^2 + b^2 = 1$.

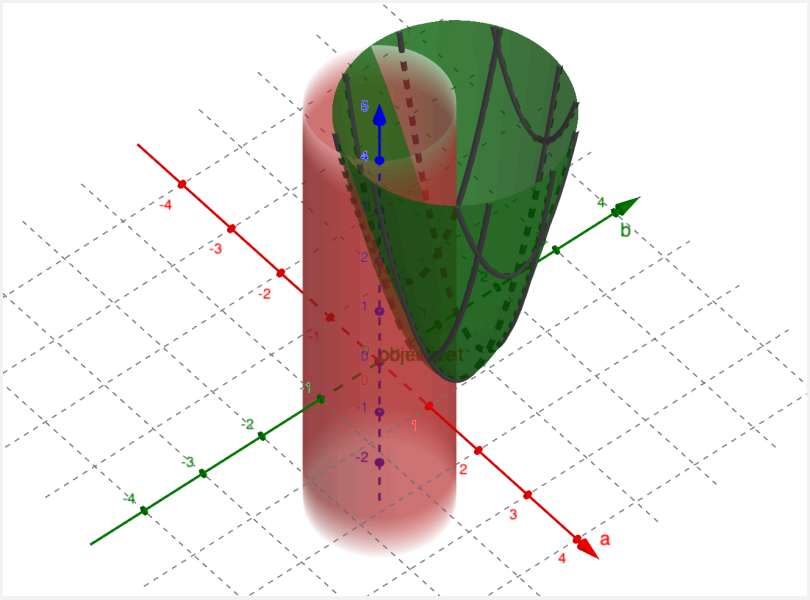


Figure 4.4 - The x and y axes of this plot correspond to the parameters a, and b, respectively, that define the 2D rotation matrix, R. The green quadratic bowl is the objective $\sum_i \|R\,{}^{O}p^{m_{c_i}} - p^{s_i}\|^2$, and the red open cylinder is the rotation matrix constraint. Click here for the interactive version.

Is that what you expected? I generated this plot using just two model/scene points, but adding more will only change the shape of the quadratic, but not its minimum. And on the interactive version of the plot, I've added a slider so that you control the parameter, θ, that represents the ground truth rotation described by R. Try it out!

In the case with no noise in the measurements (e.g. the scene points are exactly the model points modified by a rotation matrix), then the minimum of the objective function already satisfies the constraint. But if you add just a little noise to the points, then this is no longer true, and the constraint starts to play an important role.

The geometry should be roughly the same in 3D, though clearly in much higher dimensions. But I hope that the plot makes it perhaps a little less surprising that this problem has an elegant numerical solution based on the singular value decomposition (SVD).

What about translations? There is a super important insight that allows us to decouple the optimization of rotations from the optimization of translations. The insight is this: *the **relative** position between points is affected by rotation, but not by translation*. Therefore, if we write the points relative to some canonical point on the object, we can solve for rotations alone. Once we have the rotation, then we can back out the translations easily. For our least squares objective, there is even a "correct" canonical point to use -- the *central point* (like the center of mass if all points have equal mass) under the Euclidean metric.

Therefore, to obtain the solution to the problem,

$$\min_{p\in\mathbb{R}^3, R\in\mathbb{R}^{3\times 3}} \quad \sum_{i=1}^{N_s} \|p + R\,{}^{O}p^{m_{c_i}} - p^{s_i}\|^2, \tag{2}$$

$$\text{subject to} \quad RR^T = I, \tag{3}$$

first we define the central model and scene points, $\bar{m}$ and $\bar{s}$, with positions given by

$$ {}^{O}p^{\bar{m}} = \frac{1}{N_s}\sum_{i=1}^{N_s} {}^{O}p^{m_{c_i}}, \qquad p^{\bar{s}} = \frac{1}{N_s}\sum_{i=1}^{N_s} p^{s_i}. $$

If you are curious, these come from taking the gradient of the Lagrangian with respect to p, and setting it equal to zero:

$$ \sum_{i=1}^{N_s} 2(p + R\,{}^{O}p^{m_{c_i}} - p^{s_i}) = 0 \Rightarrow p^* = \frac{1}{N_s}\sum_{i=1}^{N_s} p^{s_i} - R^*\left(\frac{1}{N_s}\sum_{i=1}^{N_s} {}^{O}p^{m_{c_i}}\right), $$

but don't forget the geometric interpretation above. This is just another way to see that we can substitute the right-hand side in for p in our objective and solve for R by itself.

To find R, compose the data matrix

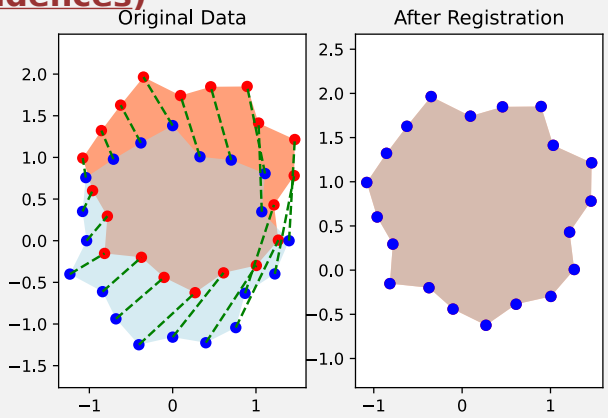
$$ W = \sum_{i=1}^{N_s} (p^{s_i} - p^{\bar{s}})({}^{O}p^{m_{c_i}} - {}^{O}p^{\bar{m}})^T, $$

and use SVD to compute $W = U\Sigma V^T$. The optimal solution is

$$ R^* = UDV^T, $$
$$ p^* = p^{\bar{s}} - R^{*\,O}p^{\bar{m}}, $$

where D is the diagonal matrix with entries $[1, 1, \det(UV^T)]$ [5]. This may seem like magic, but replacing the singular values in SVD with ones gives the optimal projection back onto the orthonormal matrices, and the diagonal matrix ensures that we do not get an improper rotation. There many derivations available in the literature, but many of them drop the determinant constraint instead of handling the improper rotation. See [6] (section 3) for one of my early favorites.

Example 4.3 (Point cloud registration with known correspondences)



In the example code, I've made a random object (based on a random set of points in the x-y plane), and perturbed it by a random 2D transform. The blue points are the model points in the model coordinates, and the red points are the scene points. The green dashed lines represent the (perfect) correspondences. On the right, I've plotted both points again, but this time using the estimated pose to put them both in the world frame. As expected, the algorithm perfectly recovers the ground truth transformation.

What is important to understand is that once the correspondences are given we have an efficient and robust numerical solution to estimating the pose.

4.4 Iterative Closest Point (ICP)

So how do we get the correspondences? In fact, if we were given the pose of the object, then figuring out the correspondences is actually easy: the corresponding point on the model is just the nearest neighbor / closest point to the scene point when they are transformed into a common frame.

This observation suggests a natural iterative scheme, where we start with some initial guess of the object pose and compute the correspondences via closest points, then use those correspondences to update the estimated pose. This is one of the famous and often used (despite its well-known limitations) algorithms for point cloud registration: the iterative closest point (ICP) algorithm.

To be precise, let's use $\hat{X}^O$ to denote our estimate of the object pose, and $\hat{c}$ to denote our estimated correspondences. The "closest-point" step is given by

$$\forall i, \hat{c}_i = \text{argmin}_{j \in N_m} \|\hat{X}^O \, {}^O p^{m_j} - p^{s_i}\|^2.$$

In words, we want to find the point in the model that is the closest in Euclidean distance to the transformed scene points. This is the famous "nearest neighbor" problem, and we have good numerical solutions (using optimized data structures) for it. For instance, Open3D uses FLANN[7].

Although solving for the pose and the correspondences jointly is very difficult, ICP leverages the idea that if we solve for them independently, then both parts have good solutions. Iterative algorithms like this are a common approach taken in optimization for e.g. bilinear optimization or expectation maximization. It is important to understand that this is a local solution to a non-convex optimization problem. So it is subject to getting stuck in local minima.

Example 4.4 (Iterative Closest Point)

Click here for the animation.

Here is ICP running on the random 2D objects. Blue are the model points, red are the scene points, green dashed lines are the correspondences. I encourage you to run the code yourself.

I've included one of the animation where it happens to converge to the true optimal solution. But it gets stuck in a local minima more often than not! I hope that stepping through will help your intuition. Remember that once the correspondences are correct, the pose estimation will recover the exact solution. So every one of those steps before convergence has at least a few bad correspondences. Look closely!

Intuition about these local minima has motivated a number of ICP variants, including point-to-plane ICP, normal ICP, ICP that use color information, feature-based ICP, etc. A particular variant based on "point-pair features" has been highly effective in a (nearly) annual object pose estimation challenge[8, 9].

4.5 Dealing with partial views and outliers

The example above is sufficient to demonstrate the problems that ICP can have with local minima. But we have so far been dealing with unreasonably perfect point clouds. Point cloud registration is an important topic in many fields, and not all of the approaches you will find in the literature are well suited to dealing with the messy point clouds we have to deal with in robotics.

Example 4.5 (ICP with messy point clouds)

I've added a number of parameters for you to play with in the notebook to add outliers (scene points that do not actually correspond to any model point), to add noise, and/or to restrict the points to a partial view. Please try them by themselves and in combination.

Figure 4.7 - A sample run of ICP with outliers (modeled as additional points drawn from a uniform distribution over the workspace).

Click here for the animation.

Figure 4.8 - A sample run of ICP with Gaussian noise added to the positions of the scene points.

Click here for the animation.

Figure 4.9 - A sample run of ICP with scene points limited to a partial view. They aren't all this bad, but when I saw that one, I couldn't resist using it as the example!

4.5.1 Detecting outliers

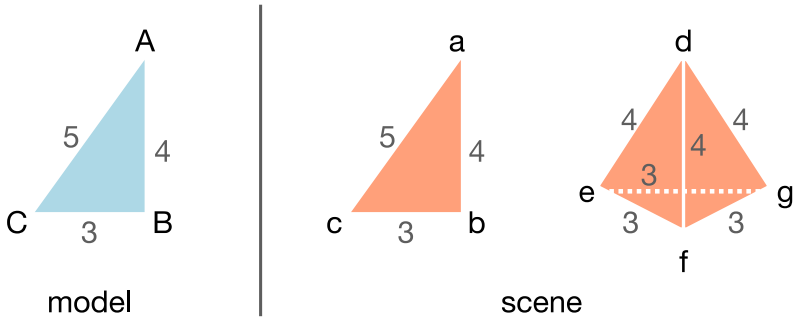
Once we have a reasonable estimate, X^O, then detecting outliers is straight-forward. Each scene point contributes a term to the objective function according to the (squared) distance between the scene point and its corresponding model point. Scene points that have a large distance can be labeled as outliers and removed from the point cloud, allowing us to refine the pose estimate without these distractors. You will find that many ICP implementations, such as this one from Open3d include a "maximum correspondence distance" parameter for this purpose.

This leaves us with a "chicken or the egg" problem -- we need a reasonable estimate of the pose to detect outliers, but for very messy point clouds it may be hard to get a reasonable estimate in the presence of those outliers. So where do we begin? One common approach that was traditionally used with ICP is an algorithm called RANdom SAmple Consensus (RANSAC) [10]. In RANSAC, we select a number of random (typically quite small) subsets of "seed" points, and compute a pose estimate for each. The subset with the largest number of "inliers" -- points that are below the maximum correspondence distance -- is considered the winning set and can be used to start the ICP algorithm.

There is another important and clever idea to know. Remember the important observation that we can decouple the solution for rotation and translation by exploiting the fact that the *relative* position between points depends on the rotation but is invariant to translation. We can take that idea one step further and note that the *relative* ***distance*** between points is actually invariant to both rotation *and* translation. This has traditionally been used to separate the estimation of scale from the estimation of translation and rotation (I've decided to ignore scaling throughout this chapter, since it doesn't make immediate sense for our proposed problem). In [11] they proposed that this can also be used to prune outliers even before we have an initial pose estimate. If the distance between a *pair* of points in the scene is unlike any distance between any pair of points in the model, then one of those two points is an outlier. Finding the maximum set of inliers by this metric would correspond to finding the maximum clique in a graph; one can make strong assumptions to guarantee that the maximum clique is the best match, or simply admit that this could be a useful heuristic. If finding maximum cliques is too expensive for large point clouds, conservative approximations may be used instead.

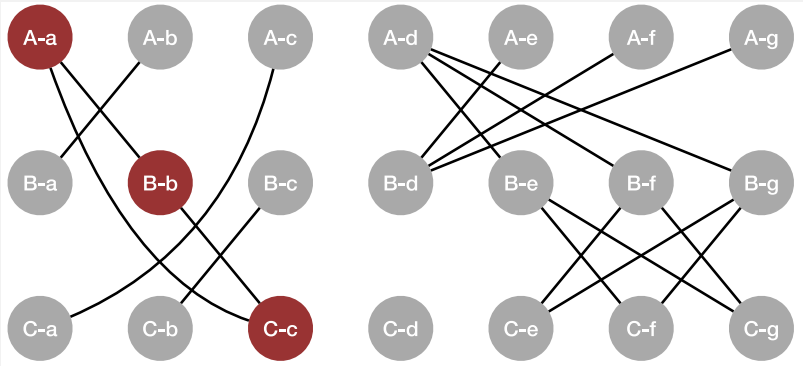
Example 4.6 (Maximum cliques in correspondence graphs)

Here is a simple example of the maximum-clique idea from [11] that I worked through with the author, Hank, to make sure I understood his idea.



The model (on the left) is a simple triangle with vertex points labeled $[A, B, C]$. The scene (on the right) has the same triangle, with points labeled $[a, b, c]$ which will have exactly three pairwise distances. But it also has a pyramid that will create a total of six pairwise distance matches (three of length 3 and three of length 4). Will the spurious correspondences on the pyramid cause a larger clique that the ground truth correspondences $[A-a, B-b, C-c]$ triangle?

Let's construct the pairwise distance graph using all-to-all correspondences as the initial guess. We'll get a node in that graph for every possible correspondence, e.g. I've used the label $A-e$ to denote a possible correspondence of model point A with scene point e. Then we make an edge in the graph between $(A-e)$ and $(B-f)$ iff the distance between A and B equals the distance between e and f.



Now the value of checking for a clique becomes clear. The largest clique involving correspondences with the pyramid is only size 2, but the largest clique for the scene triangle is size 3 (I've colored it red). So for this example, the maximum clique *does*, in fact, recover the ground-truth correspondences.

These are two examples of a zoo of algorithms / heuristics for removing outliers. One more example to mention is [8], which uses an optimized voting/clustering scheme based on the point-pair features, which they have made highly efficient, and which has performed very well for years in the online pose estimation benchmarks.

4.5.2 Point cloud segmentation

Not all outliers are due to bad returns from the depth camera, very often they are due to other objects in the scene. Even in the most basic case, our object of interest

will be resting on a table or in a bin. If we know the geometry of the table/bin a priori, then we can subtract away points from that region. Alternatively, we can `Crop` the point cloud to a region of interest. This can be sufficient for very simple or uncluttered scenes, and will be just enough for us to accomplish our goal for this chapter.

More generally, if we have multiple objects in the scene, we may still want to find the object of interest (mustard bottle?). If we had truly robust point cloud registration algorithms, then one could imagine that the optimal registration would allow us to correspond the model points with just a subset of the scene points; point cloud registration could solve the object detection problem! Unfortunately, our algorithms aren't strong enough.

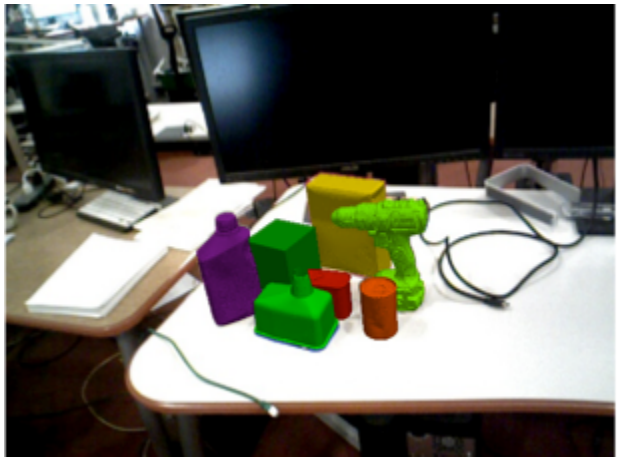


Figure 4.12 - A multi-object scene from LabelFusion [12].

As a result, we almost always use some other algorithm to "segment" the scene in a number of possible objects, and run registration independently on each segment. There are numerous geometric approaches to segmentation[13]. But these days we tend to use neural networks for segmentation, so I won't go into those details here.

4.5.3 Generalizing correspondence

The problem of outliers or multiple objects in the scene challenges our current approach to correspondences. So far, we've used the notation $c_i = j$ to represent the correspondence of *every* scene point to a model point. But note the asymmetry in our algorithm so far (scene $\Rightarrow$ model). I chose this direction because of the motivation of partial views from the mustard bottle example from the chapter opening. Once we start thinking about tables and other objects in the scene, though, maybe we should have done model $\Rightarrow$ scene? You should verify for yourself that this would be a minor change to the ICP algorithm. But what if we have multiple objects *and* partial views?

It would be simple enough to allow c_i to take a special value for "no correspondence", and indeed that helps. In that case, we would hope that, at the optimum, the scene points corresponding to the model of interest would be mapped to their respective model points, and the scene points from outliers and other objects get labelled as "no correspondence". The model points that are from the occluded parts of the object would simply not have any correspondences associated with them.

There is an alternative notation that we can use which is slightly more general. Let's denote the correspondence matrix $C \in \{0,1\}^{N_s \times N_m}$, with $C_{ij} = 1$ iff scene point i corresponds with model point j. Using this notation, we can write our estimation step (given known correspondences) as:

$$\min_{X \in \text{SE}(3)} \sum_{i=1}^{N_s} \sum_{j=1}^{N_m} C_{ij} \|X \, {}^{O}p^{m_j} - p^{s_i}\|^2.$$

This subsumes our previous approach: if $c_i = j$ then set $C_{ij} = 1$ and the rest of the row equal to zero. If the scene point i is an outlier, then set the entire row to zero. The previous notation was a more compact representation for the true asymmetry in the "closest point" computation that we did above, and made it clear that we only needed to compute N_s distances. But this more general notation will take us to the next round.

4.5.4 Soft correspondences

What if we relax our strict binary notion of correspondences, and think of them instead as correspondence weights $C_{ij} \in [0, 1]$ instead of $C_{ij} \in \{0, 1\}$? This is the main idea behind the "coherent point drift" (CPD) algorithm [14]. If you read the CPD literature you will be immersed in a probabilistic exposition using a Gaussian Mixture Model (GMM) as the representation. The probabilistic interpretation is useful, but don't let it confuse you; trust me that it's the same thing. To maximize the likelihood on a Gaussian, step one is almost always to take the log because it is a monotonic function, which gets you right back to our quadratic objective.

The probabilistic interpretation does give a natural mechanism for setting the correspondence weights on each iteration of the algorithm, by thinking of them as the probability of the scene points given the model points:

$$C_{ij} = \frac{1}{a_i} \exp \frac{- \left\| {}^{X^O} {}_{O_p} {}^{m_j} - p^{s_i} \right\|^2}{2\sigma^2}, \tag{4}$$

which is the standard density function for a Gaussian, σ is the standard deviation and a_i is the proper normalization constant to make the probabilities sum to one. It is also natural to encode the probability of an outlier in this formulation (it simply modifies the normalization constant). Click to see the normalization constant, but it's really an implementation detail. The normalization works out to be

$$a_i = (2\pi\sigma^2)^{\frac{D}{2}} \left[\sum_{j=1}^{N_m} \exp \frac{- \left\| {}^{X^O} {}_{O_p} {}^{m_j} - p^{s_i} \right\|^2}{2\sigma^2} + \frac{w}{1-w} \frac{N_m}{N_s} \right], \tag{5}$$

with $D = 3$ being the dimension of the Euclidean space and $0 \leq w \leq 1$ the probability of a sample point being an outlier [14].

The CPD algorithm is now very similar to ICP, alternating between assigning the correspondence weights and updating the pose estimate. The pose estimation step is almost identical to the procedure above, with the "central" points now

$${}^{O}p^{\bar{m}} = \frac{1}{N_C} \sum_{i,j} C_{ij} \, {}^{O}p^{m_j}, \quad p^{\bar{s}} = \frac{1}{N_C} \sum_{i,j} C_{ij} p^{s_i}, \quad N_C = \sum_{i,j} C_{ij},$$

and the data matrix now:

$$W = \sum_{i,j} C_{ij} \left(p^{s_i} - p^{\bar{s}} \right) \left({}^{O}p^{m_j} - {}^{O}p^{\bar{m}} \right)^T.$$

The rest of the updates, for extracting R using SVD and solving for p given R, are the same. You won't see the sums explicitly in the code, because each of those steps has a particularly nice matrix form if we collect the points into a matrix with one point per column.

The probabilistic interpretation also gives us a strategy for determining the covariance of the Gaussians on each iteration. [14] derives the σ estimate as: (TODO: finish converting this part of the derivation)

The word on the street is the CPD is considerably more robust than ICP with its hard correspondences and quadratic objective; the Gaussian model mitigates the effect of outliers by setting their correspondence weight to nearly zero. But it is also more expensive to compute all of the pairwise distances for large point clouds.

In [15], we point out that a small reformulation (thinking of the scene points as being distributed via a Gaussian in space, instead of a Gaussian around the model points) can get us back to summing over the scene points only. It enjoys many of the robustness benefits of CPD, but also the performance of algorithms like ICP.

4.5.5 Nonlinear optimization

All of the algorithms we've discussed so far have exploited the SVD solution to the pose estimate given correspondences, and alternate between estimating the correspondences and estimating the pose. There is another important class of algorithms that attempt to solve for both simultaneously. This makes the optimization problem nonconvex, which suggests they will still suffer from local minima like we saw in the iterative algorithms. But many authors argue that the solution times using nonlinear solvers can be on par with the iterative algorithms (e.g. [16]).

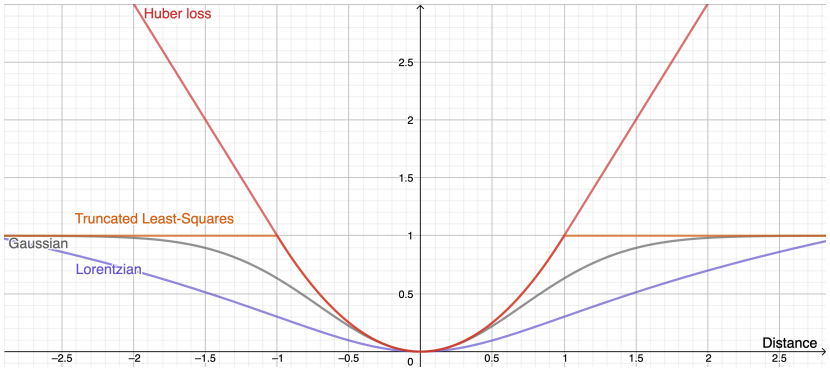


Figure 4.13 - A variety of objective functions for robust registration. Click here for the interactive version.

Another major advantage of using nonlinear optimization is that these solvers can accommodate a wide variety of objective functions to achieve the same sort of robustness to outliers that we saw from CPD. I've plotted a few of the popular objective functions above. The primary characteristic of these functions is that they taper, so that larger distances eventually do not result in higher cost. Importantly, like in CPD, this means that we can consider all pairwise distances in our objective, because if outliers add a constant value (e.g. 1) to the cost, then they have no gradient with respect to the decision variables and no impact on the optimal solution. We can therefore write, for our favorite loss function, $l(x)$, an objective of the form, e.g.

$$\min \sum_{i,j} \left[l\left(\| X^O {}^O p^{m_j} - p^{s_i} \| \right) \right].$$

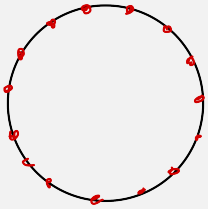
And what are the decision variables? We can also exploit the additional flexibility of the solvers to use minimal parameterizations -- e.g. θ for rotations in 2D, or Euler angles for rotations in 3D. The objective function is more complicated but we can get rid of the constraints.

Example 4.7 (Nonlinear pose estimation with known correspondences)

To understand precisely what we are giving up, let's consider a warm-up problem where we use nonlinear optimization on the minimal parameterizations for the pose estimation problem. As long as we don't add any constraints, we can still separate the solutions for rotation from the solutions from translation. So consider the problem:

$$\min_{\theta} \sum_{j} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \left({}^{O}p^{m_{c_i}} - {}^{O}p^{\bar{m}} \right) - \left(p^{s_i} - p^{\bar{s}} \right)^{2}.$$

We now have a complex, nonlinear objective function. Have we introduced local minima into the problem?



As a thought experiment, consider the problem where all of our model points lie on a perfect circle. For any one scene point i, in the rotation-only optimization, the worst case is when our estimate θ is 180 degrees (π radians) away from the optimal solution. The cost for that point would be $4r^2$, where r is the radius of the circle. In fact, using the law of cosines, we can actually write the squared distance for the point for any error, $\theta_{err} = \theta - \theta^*$, as

$$\text{distance}^2 = r^2 + r^2 - 2r^2 \cos\theta_{err}.$$

And in the case of the circle, every other model point contributes the same cost. This is not a convex function, but every minima is a globally optimal solution (just wrapped around by 2π).

In fact, even if we have a more complicated, non-circular, geometry, then this same argument holds. Every point will incur and error as it rotates around the circle, but may have a different radius. The error for all of the model points will decrease as $\cos\theta_{err}$ goes to one. So every minima is a globally optimal solution. We haven't actually introduced local minima (yet)!

It is the correspondences, and other constraints that we might add to the problem, that really introduce local minima.

Precomputing distance functions

There is an exceptionally nice trick for speeding up the computation in these nonlinear optimizations, but it does require us to go back to thinking about the minimum distance from a scene point to the model, as opposed to the pairwise distance between points. This, I think, it not a big sacrifice now that we have stopped enumerating correspondences explicitly. Let's slightly modify our objective above to be of the form

$$\min \sum_{i} \left[l\left(\min_{j} \| X^{O}\, {}^{O}p^{m_j} - p^{s_i} \| \right) \right].$$

In words, we can an apply arbitrary robust loss function, but we will only apply it to the minimum distance between the scene point and the model.

The nested $\min$ functions look a little intimidating from a computational perspective. But this form, coupled with the fact that in our application the model points are fixed, actually enables us to do some pre-computation to make the online step fast. First, let's move our optimization into the model frame by changing the inner term to

$$\min_{j} \| {}^{O}p^{m_j} - {}^{O}X^{W} p^{s_i} \|.$$

Now realize that for any 3D point x, we can precompute the minimum distance from x to any point on our model, call it $\phi(x)$. The term above is just $\phi({}^{O}X^{W}p^{s_i})$. This function of 3D space, sometimes called a distance field, and the closely related signed distance function (SDF) and level sets are a common representation in geometric modeling. And they are precisely what we need to quickly compute the cost from many scene points.

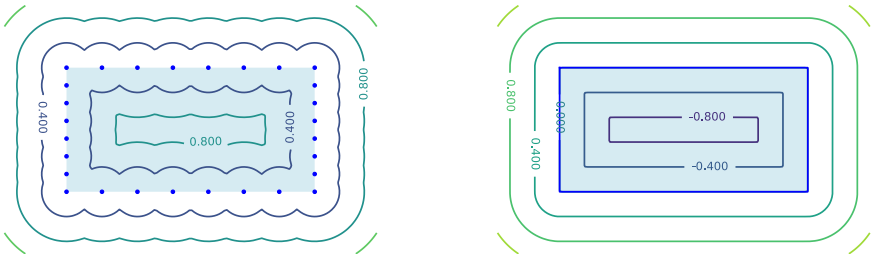


Figure 4.14 - Contour plot of the distance function for our point-based representation of the rectange (left), and the *signed* distance function for our "mesh-based" 2D rectangle. The smoother distance contours in the mesh-based representation can help alleviate some of the local minima problems we observed above.

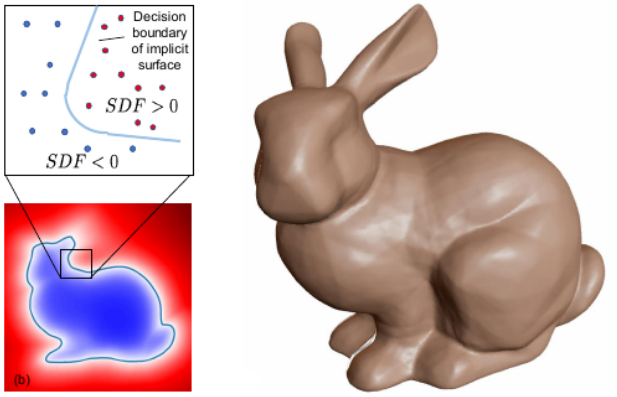


Figure 4.15 - A visualization of a "signed distance function" (SDF) representation of 3D geometry. Reproduced with permissions from [2].

As the figure I've included above suggests, we can use the precomputed distance functions to represent the minimum distance to a mesh model instead of just a point cloud. And with some care, we can define alternative distance-like functions in 3D space that have smooth subgradients which can work better with nonlinear optimization. There is a rich literature on this topic; see [17] for a nice reference.

4.5.6 Global optimization

Is there any hope of exploiting the beautiful structure that we found in the pose estimation with known correspondences with an algorithm that searches for correspondences and pose simultaneously?

There are some algorithms that claim global optimality for the ICP objective, typically using a branch and bound strategy [18, 19, 20], but we have been disappointed with the performance of these on real point clouds. We proposed another, using branch and bound to explicitly enumerate the correspondences with a truncated least-squares objective, and a tight relaxation of the SE(3) constraints[21], but this method is still limited to relatively small problems.

In recent work, TEASER [11] takes a different approach, using the observation that truncated least squares can be optimized efficiently for the case of scale and translation. For rotations, they solve a semi-definite relaxation of the truncated least-squares objective. This relaxation is not guaranteed to be tight, but (like all of the convex relaxations we describe in this chapter), it is easy to certify after the fact whether the optimal solution satisfied the original constraints.

4.6 Non-penetration and "free-space" constraints

We've explored two main lineages of algorithms for the pose estimation -- one based on the beautiful SVD solutions and the other based on nonlinear optimization. As we will see, non-penetration and free-space constraints are, in most cases, non-convex constraints, so are a better match for the nonlinear optimization approach. But there are some examples of convex non-penetration constraints (e.g., when points must not penetrate a half-plane) and it *is* possible to include these in our convex optimization approach. I'll illustrate both versions here with a simple example.

Example 4.8 (Non-penetration via nonlinear optimization)

For simplicity, I will restrict this example to 2D, where we can parameterize the rotation matrices:

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

and will solve the problem with known correspondences. Please do remember, though, that we can also include the correspondence search in the nonlinear optimization framework.

Let's solve the following optimization:

$$\begin{aligned} \min_{p,\theta} \quad & \sum_i \|p + R(\theta)\, {}^O p^{m_{c_i}} - p^{s_i}\|^2, \\ \text{subject to} \quad & {}^W p^{m_i} \geq 0, \quad \forall i \in [1, N_m]. \end{aligned}$$

In order to add the nonlinear costs and constraints, we use pass a python function and the decision variables to be bound with that function to the `MathematicalProgram` `AddCost` and `AddConstraint` methods. I've provided an implementation in this notebook:

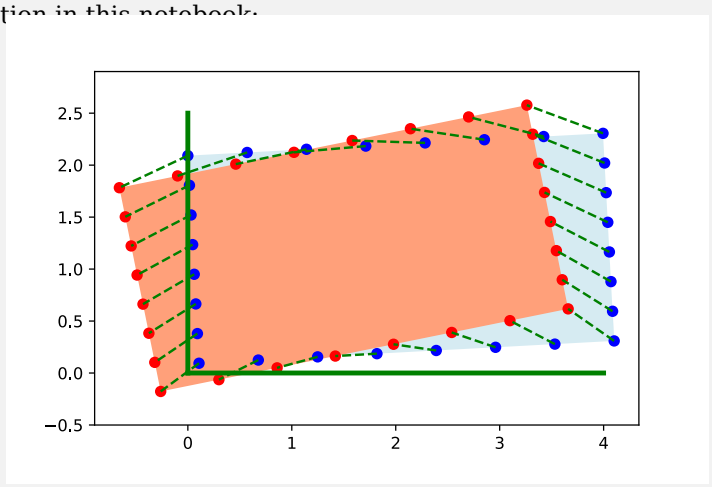


Figure 4.16 - The solution to the constrained pose estimation problem with known correspondences. I have added constraints that both the x and y positions of all of the points in the estimated pose must be greater than 0 in the world frame (as illustrated by the green lines). The red scene points are in penetration, but the solver returns the pose illusutrated by the blue, which satisfies the constraints.

Example 4.9 (Non-penetration via convex optimization)

For a convex parameterization of the rotation matrices in this 2D example, we can

leverage our convenient form of the rotation matrices:

$$R = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}.$$

Recall that the rotation matrix constraints for this form reduce to $a^2 + b^2 = 1$. These are non-convex constraints, but we can relax them to $a^2 + b^2 \leq 1$. This is almost exactly the problem we visualized in this earlier example, except that we will add translations here. The relaxation is exactly changing the non-convex unit circle constraint into the convex unit disk constraint. Based on that visualization, we have reason to be optimistic that the unit disk relaxation could be tight.

Let's solve the following optimization:

$$\begin{aligned} \min_{p,a,b} \quad & \sum_i \|p + R\, {}^O p^{m_{c_i}} - p^{s_i}\|^2, \\ \text{subject to} \quad & a^2 + b^2 \leq 1, \\ & {}^W p^{m_i} \geq 0, \quad \forall i \in [1, N_m], \end{aligned}$$

where R depends on a, b as above. Just as the addition of constraints forced us to move from the pseudo-inverse solution to a numerical optimization-based solution in the last chapter, the solution to this problem is no longer given simply by SVD. As formulated, this optimization falls under the category of a Second-Order Cone Program (SOCP), though we need to use a slack variable to put it into the standard form with a linear objective.

Please be careful. Now that we have a constraint that depends on p, our original approach to solving for R independently is no longer valid (at least not without modification). I've provided a simple implementation in the notebook.

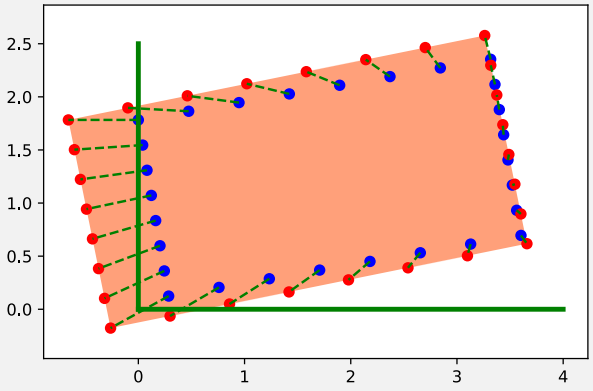


Figure 4.17 - The convex approximation to the constrained pose estimation problem with known correspondences. I have added constraints that both the x and y positions of all of the points in the estimated pose must be greater than 0 in the world frame (as illustrated by the green lines).

This is a slightly exaggerated case, where the scene points are really pulling the box into the constraints. We can see that the relaxation is *not* tight in this situation; the box is being shrunk slightly to explain the data.

Constrained optimizations like this can be made relatively efficient, and are suitable for use in an ICP loop for modest-sized point clouds. But the major limitation is that we can only take this approach for convex non-penetration constraints (or convex approximations of the constraints), like the "half-plane" constraints we used here. It is probably not very suitable for non-penetration between two objects in the scene.

For the record, the fact that I was able to easily add this example here is actually pretty special. I don't know of another toolbox that brings together the advanced optimization algorithms with the geometry / physics / dynamical systems in the way that Drake does.

4.6.1 Free space constraints as non-penetration constraints

There is another beautiful idea that I first saw in [22]...

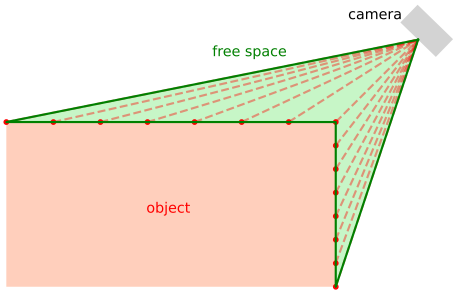


Figure 4.18 - The rays cast from the camera to the depth returns define a "free space obstacle" that other objects should not penetrate.

Ultimately, as much as I prefer the convex formulations with their potential for deeper understanding (by us) and for stronger guarantees (for the algorithms), the ability to add non-penetration constraints and free-space constraints is simply too valuable to ignore. Deep learning methods can now often provide a good initial guess, which can mitigate some of the concerns about local minima. I hope that, if you come back to these notes in a year or two, I will be able to report that we have strong results for these more complex formulations. But for now, in most applications, I will steer you towards the nonlinear optimization approaches and taking advantage of these constraints.

4.7 Tracking

4.8 Putting it all together

Now we can put all of the pieces together. In the notebook, I've created an example with the mustard bottle in one bin. First, I use ICP to localize its pose. Then, I plan a simple trajectory like we did in the last chapter to pick it up and place it in the second bin. Finally, I use differential inverse kinematics to execute it. Very satisfying!

4.9 Looking ahead

Although I very much appreciate the ability to think rigorously about these geometric approaches, we need to think critically about whether we are solving for the right objectives.

Even if we stay in the realm of pure geometry, it is not clear that a least-squares objective with equal weights on the points is correct. Imagine a tall skinny book laying flat on the table -- we might only get a very small number of returns from the

edges of the book, but those returns contain proportionally much more information than the slew of returns we might get from the book cover. It is no problem to include relative weights in the estimation objectives we have formulated here, but we don't yet have very successful geometry-based algorithms for deciding what those weights should be in any general way. (There has been a lot of research in this direction, but it's a hard problem.)

Please realize, though, that as beautiful as geometry is, we are so far all but ignoring the most important information that we have: the color values! While it is possible to put color and other features into an ICP-style pipeline, it is very hard to write a simple distance metric in color space (the raw color values for a single object might be very different in different lighting conditions, and the color values of different objects can look very similar). Advances in computer vision, especially those based on deep learning over the last few years, have brought new life to this problem. When I asked my students recently "If you had to give up one of the channels, either depth or color, which would you give up?" the answer was a resounding "I'd give up depth; don't take away my color!" That's a big change from just a few years ago. As recently as 2019, in the Benchmark for 6D Object Pose Estimation (a nearly annual competition), geometric pose estimation was still outperforming deep-learning based approaches[9]. But by the 2020 version of the challenge, deep learning had caught up.

But deep learning and geometry can (should?) work nicely together. The winners in the 2020 version of the object pose estimation challenge used deep learning to make the initial guess at the pose, but still used geometry perception (a variant of ICP) and the depth channel for refining the estimate[9].For another example, identifying correspondences between point clouds has been a major theme in this chapter -- and we haven't completely solved it. This is one of the many places where the color values and deep learning have a lot to offer [23]. We'll turn our attention to them soon!

4.10 Exercises

Exercise 4.1 (How many points do you need?)

Consider the problem of point cloud registration with known correspondences. I said that, in most cases, we have far more points than we have decision variables, and therefore treating each as an equality constraint would make the problem over-constrained. That raises a natural question:

a. What is the minimal number of points required to uniquely specify a pose in 2d? Provide a brief mathematical justification.
b. What is the minimal number of points required to uniquely specify a pose in 3d? Provide a brief mathematical justification.

Exercise 4.2 (Rotational symmetries)

We have a beautiful formulation for the point cloud registration problem with known correspondences -- it has a quadratic objective and (with the determinant constraint relaxed) a quadratic equality constraint. Sometimes we have objects that are rotationally symmetric -- think of box that is a perfect cube. How can the quadratic objective capture a problem where there should be equally good solutions at rotations that are 90 degrees apart?

Exercise 4.3 (ICP with random initializations)

If you run the ICP example code which has randomized object geometry and

ground truth object pose, you can't help but notice that the algorithm does a quite reasonable job of estimating translation, but (at least for the geometries I've generated here) does a fairly lousy job with rotation. One mitigation is to run ICP multiple times with different initial estimates of the rotation. This is a reasonable strategy for any nonconvex optimization problem with local minima, but is particularly useful here since even when the point clouds are quite complex, the dimensionality of the search space is low. In particular, we can generate a reasonable coverage of 2D or even 3D rotations with a modest number of samples (for 3D, consider using `UniformlyRandomRotationMatrix`). Furthermore, running ICP from multiple initial conditions can be done in parallel.

Try implementing ICP from multiple initial estimates $\hat{X}^O$, sampling only in rotation. If you only keep the best one (lowest estimation error), then how much does it improve the performance over a single ICP run?

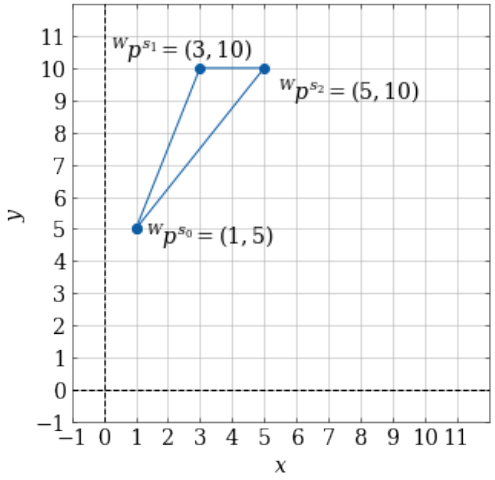
Exercise 4.4 (Point Registration with Fixed Rotation)

Consider the case of point registration where the rotation component of X^O is known, but not the translation. (In other words, the opposite of Example 4.2.)

Specifically, say your scene points ${}^W X^{CC} p^{s_i} = {}^W p^{s_i}$ are defined as follows:

$$
\begin{aligned}
{}^W p^{s_0} &= (1, 5) \\
{}^W p^{s_1} &= (3, 10) \\
{}^W p^{s_2} &= (5, 10)
\end{aligned}
$$

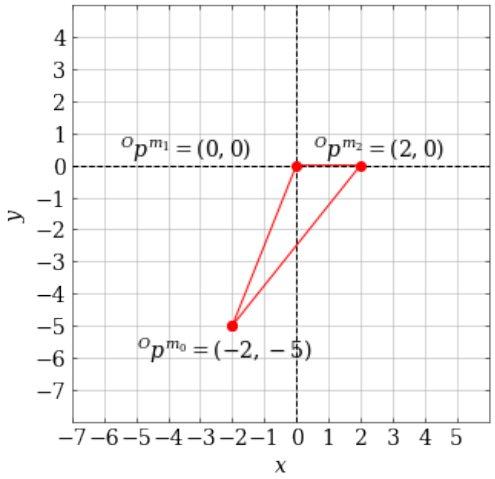
Which can be plotted as follows:



And your model points are defined as follows:

$$
\begin{aligned}
{}^O p^{m_0} &= (-2, -5) \\
{}^O p^{m_1} &= (0, 0) \\
{}^O p^{m_2} &= (2, 0)
\end{aligned}
$$

Which can be plotted as follows:



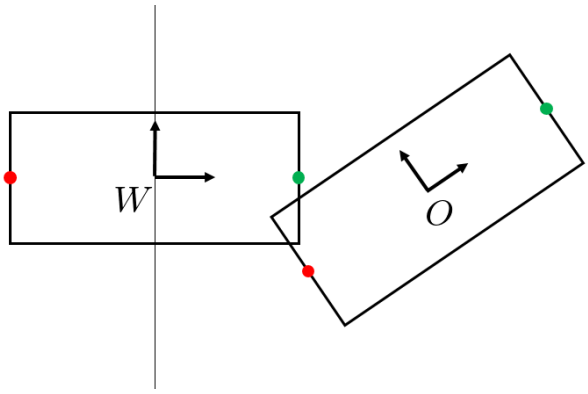
As you can see, both triangles are in the same orientation, so $R^O = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. However, we still need to solve for the translation component of X^O.

(In this example, we can easily separate translation and rotation because the orientation of the model already matches the scene, but this decoupling actually works in the general case as well. See the explanation following Example 4.2 in the textbook.)

a. What are the decision variables in this optimization problem? Be specific.
b. How do the decision variables show up in the constraints?
c. What is the value of the objective function for $p^O = (0, 0)$? What about (3,10)? And (6,12)?
d. If you plotted the objective function and constraints as a function of your decision variables, what shape would it be? Explain the significance of this.

Exercise 4.5 (Planar Two-Point ICP)

We saw that ICP has many local minimas. But what are these local minimas, and can we say something about how wrong our initial guess has to be until we arrive at one of these local minima? Although the analysis can get complicated for general geometry, let's start by analyzing a simple example of a planar two-point ICP.



The model points and the scene points are given as:

$$p^{m_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad p^{m_2} = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \qquad p^{s_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad p^{s_2} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}.$$

The true correspondence is given by their numbering, but note that we don't know the true correspondence - ICP simply determines it based on nearest neighbors. Given our vanilla-ICP cost (sum of pairwise distances squared), we can parametrize the 2D pose X^O with p_x, p_y, a, b and write down the resulting optimization as:

$$\begin{aligned} \min_{p_x, p_y, a, b} \quad & \sum_i \left\| \begin{pmatrix} p_x \\ p_y \end{pmatrix} + \begin{pmatrix} a & -b \\ b & a \end{pmatrix} p^{m_{c_i}} - p^{s_i} \right\|^2 \\ \text{s.t.} \quad & a^2 + b^2 = 1. \end{aligned}$$

a. When the initial guess for the pose results in the correct correspondences based on nearest-neighbors, show that the ICP cost is minimum when $p_x, p_y, b = 0$ and $a = 1$. Describe the set of initial poses that results in convergence to the true solution.
b. When the initial guess results in a flipped correspondence, show that ICP cost is minimum when $p_x, p_y, b = 0$ and $a = -1$. Describe the set of initial poses that results in convergence to this incorrect solution.
c. Remember that correspondence need not be one-to-one. In fact they are often not when computed based on nearest-neighbors. By constructing the data matrix W, show that when both scene points correspond to one model point, ICP gets stuck and does not achieve zero-cost. (HINT: You may assume that doing SVD on a zero matrix leads to identity matrices for U and V).

Exercise 4.6 (Bunny ICP)

For this exercise, you will implement the ICP algorithm to match pointclouds of two Stanford bunnies. You will work exclusively in . You will be asked to complete the following steps:

a. Implement a method to compute the least-squares transform given the correspondences.
b. Implement the ICP algorithm using the least square transform method from part a.

Exercise 4.7 (RANSAC)

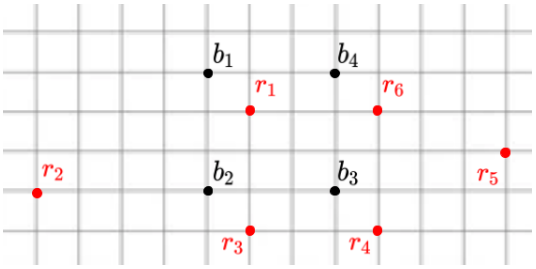
For this exercise, you will remove the environmental background from the Stanford bunny pointcloud using the RANSAC algorithm. You will work exclusively in . You will be asked to complete the following steps:

a. Implement the RANSAC algorithm.
b. Use the RANSAC algorithm to remove the planar surface from the scene point cloud.

Exercise 4.8 (Outliers in ICP)

In this question you'll explore a technique for handling outliers in ICP and think through how the distance metric impacts the robustness of ICP to outliers. There are two parts to this question and each part has three subquestions.

a. Consider the setting visualized below. Here the model points are given in black and labeled as b_i and the scene points are given in red and labeled as r_j.



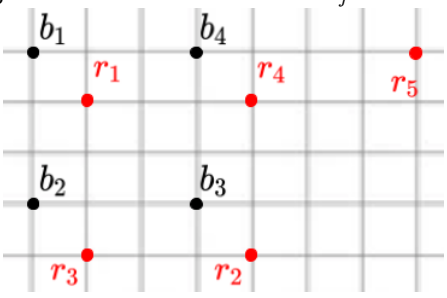
Let's consider a possible algorithm for handling the outliers in the scene.

1. First, write down each of the correspondence, i.e the (b_i, r_j) pairs. Next, compute the ICP error, the sum of the pairwise distances between each of these correspondences.
2. Suppose that you are told that $\frac{1}{3}$ of the scene points are outliers. Using the pairwise distances between each b_i and r_j, which two scene points r_j could you detect as the most likely outliers?
3. Discarding the two proposed outlier scene points, compute the new ICP error.

This is the basic intuition behind the Trimmed ICP algorithm!

b. The optimization in ICP captures the distance between two sets of points as the sum of pairwise Euclidean distances. As a thought experiment: another possible distance metric (which isn't as optimization friendly) is the one-way Hausdorff distance. Stated simply, given that an adversary picks a point on one shape, the one-way Hausdorff distance is the distance you are forced to travel to get to the closest point on the other shape. We could consider two schemes:
Scheme 1) The adversary picks the scene point and thus you pick the model point that is closest to this scene point.
Scheme 2) The adversary picks the model point and thus you pick the scene point that is closest to this model point.

Which is more robust to outliers? Let's explore through a 2D example. Again, the model points are given in black and labeled as b_i and the scene points are given in red and labeled as r_j.



1. Lets consider scheme 1. I, as the adversary, pick the scene point r_5. Whats the nearest model point b_i you can pick? Whats the distance to this model point?
2. Now in scheme 2, I pick the model point b_1. What's the nearest scene point r_j that you can pick? What's the distance to this scene point?
3. From this, which scheme was more robust to the outlier?

Exercise 4.9 (Pose Estimation)

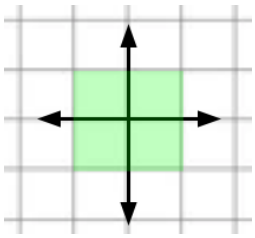
Let's go back to the example from Chapter 3, but relax the assumption that we have the pose of the red foam brick. You will be given a simulated raw pointcloud from our previous setup from a depth camera. Your task is to perform segmentation on this raw pointcloud data, and perform ICP to estimate the pose of the brick. You will work exclusively in . You will be asked to complete the following steps:

a. Perform segmentation on the raw pointcloud to remove the background.
b. Use our class ICP implementation to correctly estimate the pose of the red foam brick.

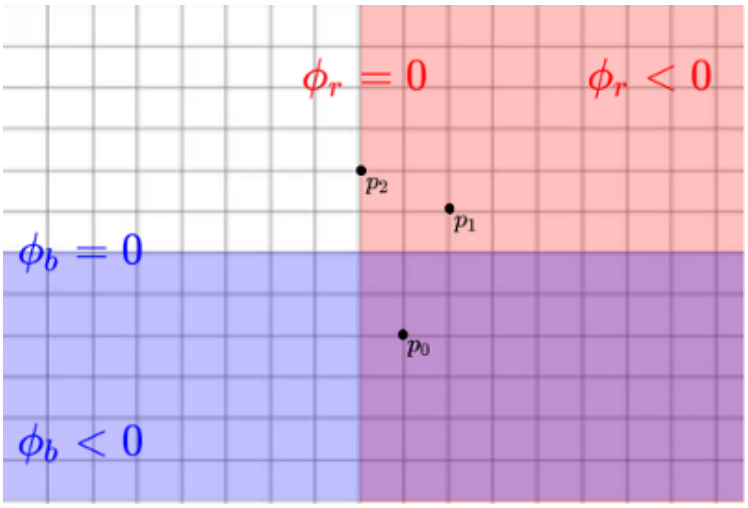
Exercise 4.10 (Composing Signed Distance Functions (SDFs))

In this problem we will explore signed distance functions and their composition! As a reminder, the signed distance function of a set gives the minimum distance of a point p to the boundary of that set, where the sign of the value is based on whether the point p is inside or outside the boundary. Points that are on the interior of the boundary are negative. and points that are on the exterior of the boundary are positive.

For a 2D example, consider the green box visualized below. The center of the box, at the origin, has a signed distance value of -1, since it is distance one from any of the edges of the box and it lies on the interior of the box. Points along the perimeter of the box have a signed distance value of 0. Any points outside the box have a signed distance value of +d where d is the smallest distance between that bound and the perimeter of the box.



In the figure below we draw two objects: a shaded red object and a shaded blue object (the purple region is two objects overlapping and the objects should be treated as infinite half planes). Let $\phi_r(p)$ be the signed distance value of a point p to boundary of the red object and $\phi_b(p)$ be the signed distance value of point p to the boundary blue object. As an example $\phi_b(p_0) = -2, \phi_b(p_1) = +1$.

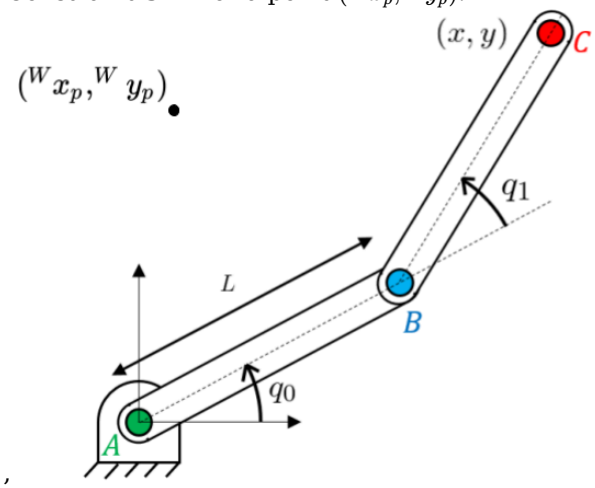


a. First, lets consider a new region defined as the union of red object and the blue object. Let $\phi_{r\cup b}(p)$ be the signed distance value of a point p to this new boundary. Compute: $\phi_{r\cup b}(p_0), \phi_{r\cup b}(p_1), \phi_{r\cup b}(p_2)$.
b. Next, lets consider a new region defined as the intersection of the red object and the blue object (hence the purple object). Let $\phi_{r\cap b}(p)$ be the signed distance value of a point p to this new boundary. Compute: $\phi_{r\cap b}(p_0), \phi_{r\cap b}(p_1), \phi_{r\cap b}(p_2)$.
c. Using your experience from part (a), describe a method for computing the signed distance value for any point p in our 2D space to the **union** of the red and blue objects. Your method should, in part, leverage the individual signed distance values of the objects. We'd encourage you to think in terms of cases based on what region(s) the point is in. Using your experience from part (b), describe an analogous method for compute the signed distance value to the **intersection** of the red and blue objects (i.e. the purple region). HINT: Write a case statement that cases on the 4 regions, then try to reduce it to 2 cases with a clean math expression.

One incorrect method to compute the SDF value for the union of two objects is to simply take the minimum of the SDF value for each of the objects. (It is worth taking a minute to convince yourself why this method is incorrect!). While this method can produce the incorrect distance, it will produce a value with the correct sign (again, convince yourself this is true!).
Therefore, this "incorrect" method can tell us if we are in penetration or out of penetration for the union of our set of objects. This makes it useful for computing a non-penetration constraint! We'll refer to the SDF generated by this "incorrect-but-useful" method as Penetration-Constraint-SDF.
Let's see how we can use this!

d. As discussed in class, for rigid objects, we can precompute the SDF for the object, which enables us to greatly speed up computations. However, if we are dealing with an articulated body, such as a robot arm, where the configuration of the arm is dependent on the joint values, we can no longer precompute the SDF. Recomputing this "global" SDF every time the robot's configuration changes gets very expensive. (By global we mean an SDF of the entire articulated body.)
Instead, the DART (Dense Articulated Real-Time Tracking) [22] algorithm considers that for each link in the articulated body, we have a "local" SDF. A local SDF gives the signed distance value to a point in the link's coordinate frame. Because we are operating in the link's coordinate frame, we can precompute these local SDFs. Consider that we have our favorite, planar two-link arm, picture below. Given q_0, q_1 and the precomputed local SDFS for each of the links (link AB, link

BC), briefly describe how you could compute the value of the global Penetration-Constraint-SDF for a point $({}^W x_p, {}^W y_p)$.



,

References

1. Chris Sweeney and Greg Izatt and Russ Tedrake, "A Supervised Approach to Predicting Noise in Depth Images", *Proceedings of the IEEE International Conference on Robotics and Automation* , May, 2019. [link]

2. Jeong Joon Park and Peter Florence and Julian Straub and Richard Newcombe and Steven Lovegrove, "{DeepSDF}: Learning Continuous Signed Distance Functions for Shape Representation", *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)* , June (To Appear), 2019. [link]

3. Ben Mildenhall and Pratul P Srinivasan and Matthew Tancik and Jonathan T Barron and Ravi Ramamoorthi and Ren Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis", *Communications of the ACM*, vol. 65, no. 1, pp. 99--106, 2021.

4. Berthold K.P. Horn, "Closed-form solution of absolute orientation using unit quaternions", *Journal of the Optical Society of America A*, vol. 4, no. 4, pp. 629-642, April, 1987.

5. Andriy Myronenko and Xubo Song, "On the closed-form solution of the rotation matrix arising in computer vision problems", *arXiv preprint arXiv:0904.1613*, 2009.

6. Robert M Haralick and Hyonam Joo and Chung-Nan Lee and Xinhua Zhuang and Vinay G Vaidya and Man Bae Kim, "Pose estimation from corresponding point data", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 19, no. 6, pp. 1426--1446, 1989.

7. Marius Muja and David G Lowe, "Flann, fast library for approximate nearest neighbors", *International Conference on Computer Vision Theory and Applications (VISAPP'09)* , vol. 3, 2009.

8. Bertram Drost and Markus Ulrich and Nassir Navab and Slobodan Ilic, "Model globally, match locally: Efficient and robust 3D object recognition", *2010 IEEE computer society conference on computer vision and pattern recognition* , pp. 998--1005, 2010.

9. Tom{\'a}s Hodan and Martin Sundermeyer and Bertram Drost and Yann Labb{\'e} and Eric Brachmann and Frank Michel and Carsten Rother and Jir{\'i} Matas, "{BOP} Challenge 2020 on {6D} Object Localization", *European Conference on Computer Vision Workshops (ECCVW)*, 2020.

10. M.A. Fischler and R.C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis andd Automated Cartography", *Communications of the Association for Computing Machinery ({ACM})* , vol. 24, no. 6, pp. 381--395, 1981.

11. Heng Yang and Jingnan Shi and Luca Carlone, "Teaser: Fast and certifiable point cloud registration", *arXiv preprint arXiv:2001.07715*, 2020.

12. Pat Marion and Peter R. Florence and Lucas Manuelli and Russ Tedrake, "A Pipeline for Generating Ground Truth Labels for Real {RGBD} Data of Cluttered Scenes", *International Conference on Robotics and Automation (ICRA), Brisbane, Australia*, May, 2018. [link]

13. Anh Nguyen and Bac Le, "3D point cloud segmentation: A survey", *2013 6th IEEE conference on robotics, automation and mechatronics (RAM)* , pp. 225--230, 2013.

14. Andriy Myronenko and Xubo Song, "Point set registration: Coherent point drift", *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 12, pp. 2262--2275, 2010.

15. Wei Gao and Russ Tedrake, "FilterReg: Robust and Efficient Probabilistic Point-Set Registration using Gaussian Filter and Twist Parameterization", *Conference on Computer Vision and Pattern Recognition (CVPR)*, June, 2019. [link]

16. Andrew W Fitzgibbon, "Robust registration of 2D and 3D point sets", *Image and vision computing*, vol. 21, no. 13-14, pp. 1145--1153, 2003.

17. Stanley Osher and Ronald Fedkiw, "Level Set Methods and Dynamic Implicit Surfaces", Springer , 2003.

18. Natasha Gelfand and Niloy J Mitra and Leonidas J Guibas and Helmut Pottmann, "Robust global registration", *Symposium on geometry processing* , vol. 2, no. 3, pp. 5, 2005.

19. Nicolas Mellado and Dror Aiger and Niloy J Mitra, "Super 4pcs fast global pointcloud registration via smart indexing", *Computer Graphics Forum* , vol. 33, no. 5, pp. 205--215, 2014.

20. Jiaolong Yang and Hongdong Li and Dylan Campbell and Yunde Jia, "Go-{ICP}: A globally optimal solution to 3D ICP point-set registration", *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 11, pp. 2241--2254, 2015.

21. Gregory Izatt and Hongkai Dai and Russ Tedrake, "Globally Optimal Object Pose Estimation in Point Clouds with Mixed-Integer Programming", *International Symposium on Robotics Research*, Dec, 2017. [link]

22. Tanner Schmidt and Richard Newcombe and Dieter Fox, "DART: dense articulated real-time tracking with consumer depth cameras", *Autonomous Robots*, vol. 39, no. 3, pp. 239--258, 2015.

23. Peter R. Florence* and Lucas Manuelli* and Russ Tedrake, "Dense Object Nets: Learning Dense Visual Object Descriptors By and For Robotic Manipulation", *Conference on Robot Learning (CoRL)* , October, 2018. [link]

Chapter 5
Bin Picking

Our initial study of geometric perception gave us some powerful tools, but also revealed some major limitations. In the next chapter, we will begin applying techniques from deep learning to perception. Spoiler alert: those methods are going to be insatiable in their hunger for data. So before we get there, I'd like to take a brief excursion into a nice subproblem that might help us feed that need.

In this chapter we'll consider the simplest version of the bin picking problem: the robot has a bin full of random objects and simply needs to move those objects from one bin to the other. We'll be agnostic about what those objects are and about where they end up in the other bin, but we would like our solution to achieve a reasonable level of performance for a very wide variety of objects. This turns out to be a pretty convenient way to create a training ground for robot learning -- we can set the robot up to move objects back and forth between bins all day and night, and intermittently add and remove objects from the bin to improve diversity. Of course, it is even easier in simulation!

Bin picking has potentially very important applications in industries such as logistics, and there are significantly more refined versions of this problem. For example, we might need to pick only objects from a specific class, and/or place the objects in known position (e.g. for "packing"). But let's start with the basic case.

5.1 Generating random cluttered scenes

If our goal is to test a diversity of bin picking situations, then the first task is to figure out how to generate diverse simulations. How should we populate the bin full of objects? So far we've set up each simulation by carefully setting the initial positions (in the Context) for each of the objects, but that approach won't scale.

So far we've used robot description files (e.g., URDF, SDF, MJCF, or Model Directives) to parse into the MultibodyPlant; those were sufficient for setting up the bins, cameras, and the robot which can be welded in fixed positions. But generating distributions over objects and object initial conditions is more advanced, and we need an algorithm for that.

5.1.1 Falling things

In the real world, we would probably just dump the random objects into the bin. That's a decent strategy for simulation, too. We can roughly expect our simulation to faithfully implement multibody physics as long as our initial conditions (and time step) are reasonable; the physics isn't well defined if we initialize the Context with multiple objects occupying the same physical space. The simplest and most common way to avoid this is to generate a random number of objects in random poses, with their vertical positions staggered so that they trivially start out of penetration.

If you look for them, you can find animations of large numbers of falling objects in the demo reels for most advanced multibody simulators. (These demos are actually a bit of a gimmick; the simulations may look realistic to us, but they are often not very physically accurate.) For our purposes the falling dynamics themselves are not the focus. We just want the state of the objects where they are done falling as initial conditions for our manipulation system.

Example 5.1 (Piles of foam bricks in 2D)

Here is the 2D case. I've added many instances of our favorite red foam brick to the plant. Note that it's possible to write highly optimized 2D simulators; that's not what I've done here. Rather, I've added a planar joint connecting each brick to the world, and run our full 3D simulator. The planar joint has three degrees of freedom. I've oriented them here to be x, z, and θ to constrain the objects to the xz plane.

I've set the initial positions for each object in the Context to be uniformly distributed over the horizontal position, uniformly rotated, and staggered every 0.1m in their initial vertical position. We only have to simulate for a little more than a second for the bricks to come to rest and give us our intended "initial conditions".

Click here for the animation.

It's not really any different to do this with any random objects -- here is what it looks like when I run the same code, but replace the brick with a random draw from a few objects from the YCB dataset [1]. It somehow amuses me that we can see the central limit theorem hard at work, even when our objects are slightly ridiculous.

Example 5.2 (Filling bins with clutter)

The same technique also works in 3D. Setting uniformly random orientations in 3D requires a little more thought, but Drake supplies the method `UniformlyRandomRotationMatrix` (and also one for quaternions and roll-pitch-yaw) to do that work for us.

Please appreciate that bins are a particularly simple case for generating random scenarios. If you wanted to generate random kitchen environments, for example, then you won't be as happy with a solution that drops refrigerators, toasters, and stools from uniformly random i.i.d. poses. In those cases, authoring reasonable distributions gets much more interesting; we will revisit the topic of generative scene models [2] later in the notes.

5.1.2 Static equilibrium with frictional contact

Even in the case of bins, we should try to think critically about whether dropping objects from a height is really the best solution. Given our discussion in the last chapter about writing optimizations with non-penetration constraints, I hope you are already asking yourself: why not use those constraints again here? Let's explore that idea a bit further.

I won't dive into a full discussion of multibody dynamics nor multibody simulation, though I do have more notes available here. What is important to understand here is that the equations of motion of our multibody system are described by differential equations of the form:

$$M(q)\dot{v} + C(q,v)v = \tau_g(q) + \sum_i J_i^T(q) f^{c_i}.$$

We already understand the generalized positions, q, and velocities, v. The left side of the equation is just a generalization of "mass times acceleration", with the mass matrix, M, and the Coriolis terms C. The right hand side is the sum of the (generalized) forces, with $\tau_g(q)$ capturing the terms due to gravity, and f^{c_i} is the spatial force due to the ith contact. $J_i(q)$ is the ith "contact Jacobian" -- it is the Jacobian that maps from the generalized velocities to the spatial velocity of the ith contact frame.

Our interest here is in finding (stable) steady-state solutions to these differential equations that can serve as good initial conditions for our simulation. At steady-state we have $v = \dot{v} = 0$, and conveniently all of the terms on the left-hand side of the equations are zero. This leaves us with just the force-balance equations

$$\tau_g(q) = -\sum_i J_i^T(q) f^{c_i}.$$

But we still need to understand where the contact forces come from.

Collision geometry

Geometry engines for robotics, like `SceneGraph` in DRAKE, distinguish between a few different roles that geometry can play in a simulation. In robot description files, we distinguish between visual and collision geometries. In particular, every rigid body in the simulation can have multiple *collision* geometries associated with it (playing the "proximity" role). Collision geometries are often much simpler than the visual geometries we use for illustration and simulating perception -- sometimes they are just a low-polygon-count version of the visual mesh and sometimes we actually use much simpler geometry (like boxes, spheres, and cylinders). These simpler geometries make the physics engine faster and more robust.

For subtle reasons we will explore below, in addition to simplifying the geometry, we sometimes over-parameterize the collision geometry in order to make the numerics more robust. For example, when we simulate the red brick, we actually use *nine* collision geometries for the one body.

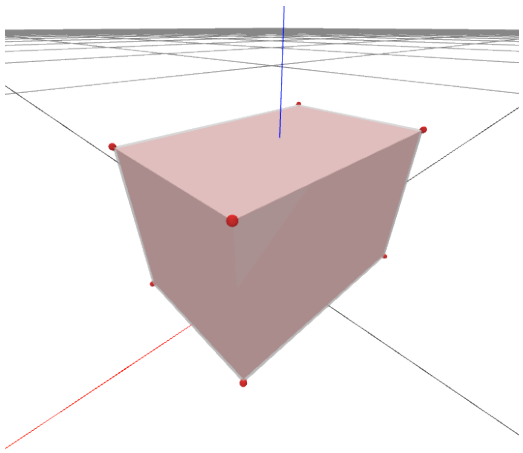


Figure 5.4 - The (exaggerated) contact geometry used for robust simulation of boxes. We add contact "points" (epsilon radius spheres) to each corner, and have a slightly inset box for the remaining contacts. Here is the interactive version.

Example 5.3 (Collision geometry inspector)

Drake's has a very useful `ModelVisualizer` tool that publishes by the illustration and collision geometry roles to Meshcat (see, for example, the Drake tutorial on "authoring a multibody simulation"). This is very useful for designing new models, but also for understanding the contact geometry of existing models.

Launch in Deepnote

SceneGraph also implements the concept of a collision filter. It can be important to specify that, for instance, the iiwa geometry at link 5 cannot collide with the geometry in links 4 or 6. Specifying that some collisions should be ignored not only speeds up the computation, but it also facilitates the use of simplified collision geometry for links. It would be extremely hard to approximate link 4 and 5 accurately with spheres, and cylinders if I had to make sure that those spheres and cylinders did not overlap in any feasible joint angle. The default collision filter settings should work fine for most applications, but you can tweak them if you like.

So where do the contact forces, f^{c_i}, come from? There is potentially an equal and opposite contact force for every *pair* of collision geometries that are not filtered out by the collision filter. In `SceneGraph`, the `GetCollisionCandidates` method returns them all. We'll take to calling the two bodies in a collision pair "body A" and "body B".

The Contact Frame

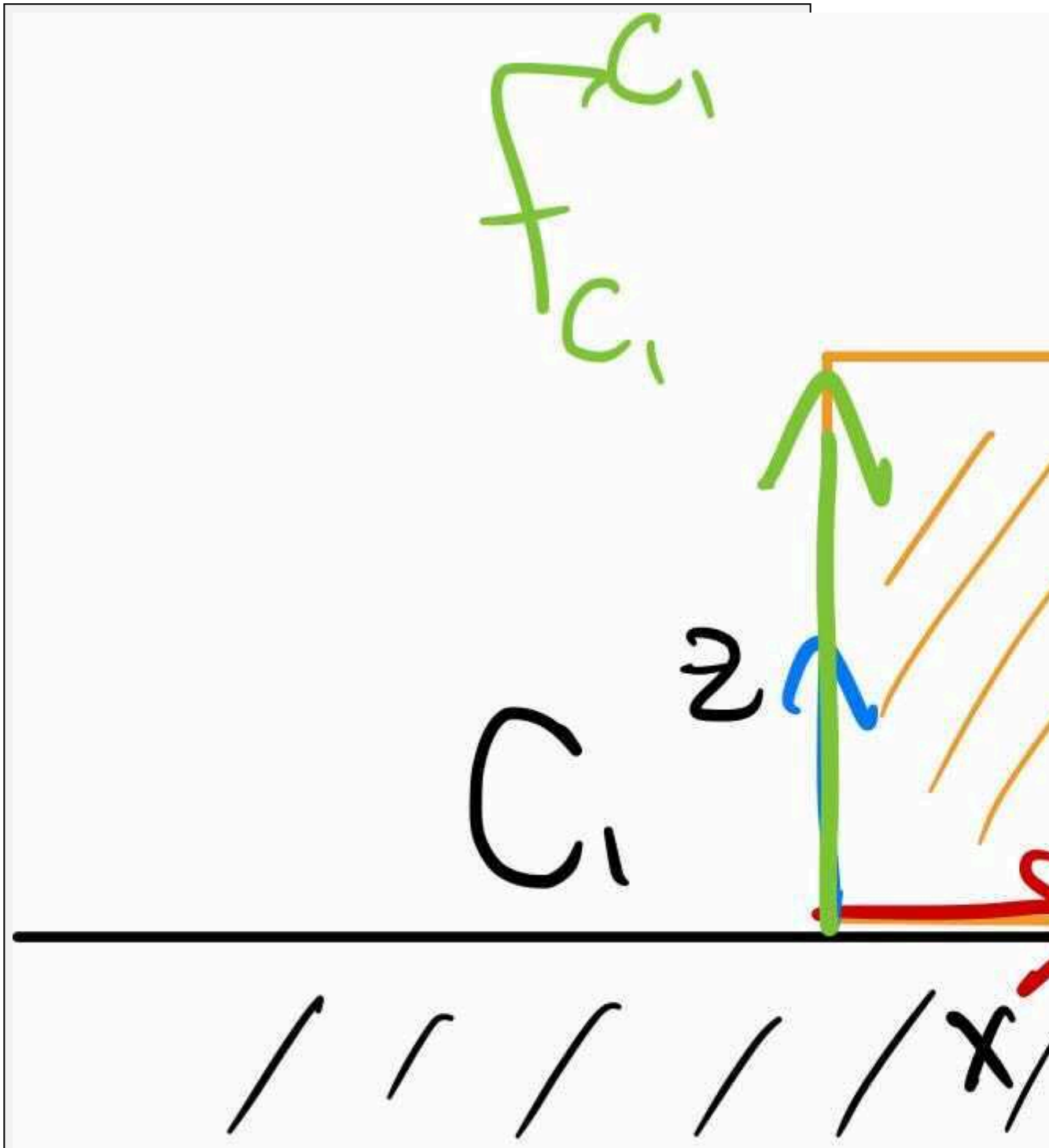
We still need to decide the magnitude and direction of these spatial forces, and to do this we need to specify the *contact frame* in which the spatial force is to be applied. For instance, we might use C_B to denote a contact frame on body B, with the forces applied at the origin of the frame.

Most simulators summarize the contact between two bodies as a translational force (e.g. zero torque) at one or more contact points. Our convention will be to align the positive z axis with the "contact normal", with positive forces resisting penetration. Defining this normal can be deceptively complicated. For instance, what is the normal at the corner of a box? Taking the normal as a (sub-)gradient of the signed-distance function for the collision geometry provides a reliable definition that will extend to the distance between two bodies and into generalized coordinates. The x and y axes of the contact frame are any orthogonal basis for the tangential coordinates. You can find additional figures and explanations here.

Example 5.4 (Brick on a half-plane)

Let's work through these details on a simple example -- our foam brick sitting on the ground. The ground is welded to the world, so has no degrees of freedom; we can ignore the forces applied to the ground and focus on the forces applied to the brick.

Where are the contact forces and contact frames? If we used only box-on-box geometry, then the locations of the contact forces are ambiguous during this "face-on-face" contact; this is even more true in 3D (where three forces would be sufficient). But by adding extra contact spheres to the corners of our box, we are telling the physics engine that we specifically want contact forces at each of the corners (and all four of the corners in 3D). I've labeled the frames C_1 and C_2 in the sketch below.


C1
C1
C1
2

In order to achieve static equilibrium, we require that all of the forces and torques on the brick be in balance. This is a simple manifestation of the equations

$$\tau_g(q) = -\sum_i J_i^T(q) f^{c_i},$$

because the configuration q is the x, z, θ position and orientation of the brick; we have three equations and three unknowns:

$$0 = \sum_i f_{C_{i,x}}^{c_i}$$
$$-mg = -\sum_i f_{C_{i,z}}^{c_i}$$
$$0 = \sum_i \left[\left[p^{C_i} - p^{B_{cm}} \right] \times f^{c_i} \right]_{W_y}.$$

The last equation represents the torque balance, taken around the center of mass of the brick which I've call $p^{B_{cm}}$. Torque about θ corresponds to the y component of the cross product. The torque balance ensures that $f_{C_{1,z}}^{c_1} = f_{C_{2,z}}^{c_2}$ assuming the center of mass is in the middle of the box; force balance in z set them equal to $\frac{mg}{2}$. We haven't said anything about the horizontal forces yet (0 is a reasonable solution here). Let's develop that next.

The (Coulomb) Friction Cone

Now the rules governing contact forces can begin to take shape. First and foremost, we demand that there is no force at a distance. Using $\phi_i(q)$ to denote the distance between two bodies in configuration q, we have

$$\phi(q) > 0 \Rightarrow f^{c_i} = 0.$$

Second, we demand that the normal force only resists penetration; bodies are never pulled into contact:

$$f_{C_z}^{c_i} \geq 0.$$

In *rigid* contact models, we solve for the smallest normal force enforces the non-penetration constraint (this is known as the principle of least constraint). In *soft* contact models, we define the force to be a function of the penetration depth and velocity.

Forces in the tangential directions are due to friction. The most commonly used model of friction is Coulomb friction, which states that

$$|f_{C_{x,y}}^{c_i}|_2 \leq \mu f_{C_z}^{c_i},$$

with μ a non-negative scalar *coefficient of friction*. Typically we define both a μ_{static}, which is applied when the tangential velocity is zero, and $\mu_{dynamic}$, applied when the tangential velocity is non-zero. In the Coulomb friction model, the tangential contact force is the force within this friction cone which produces maximum dissipation.

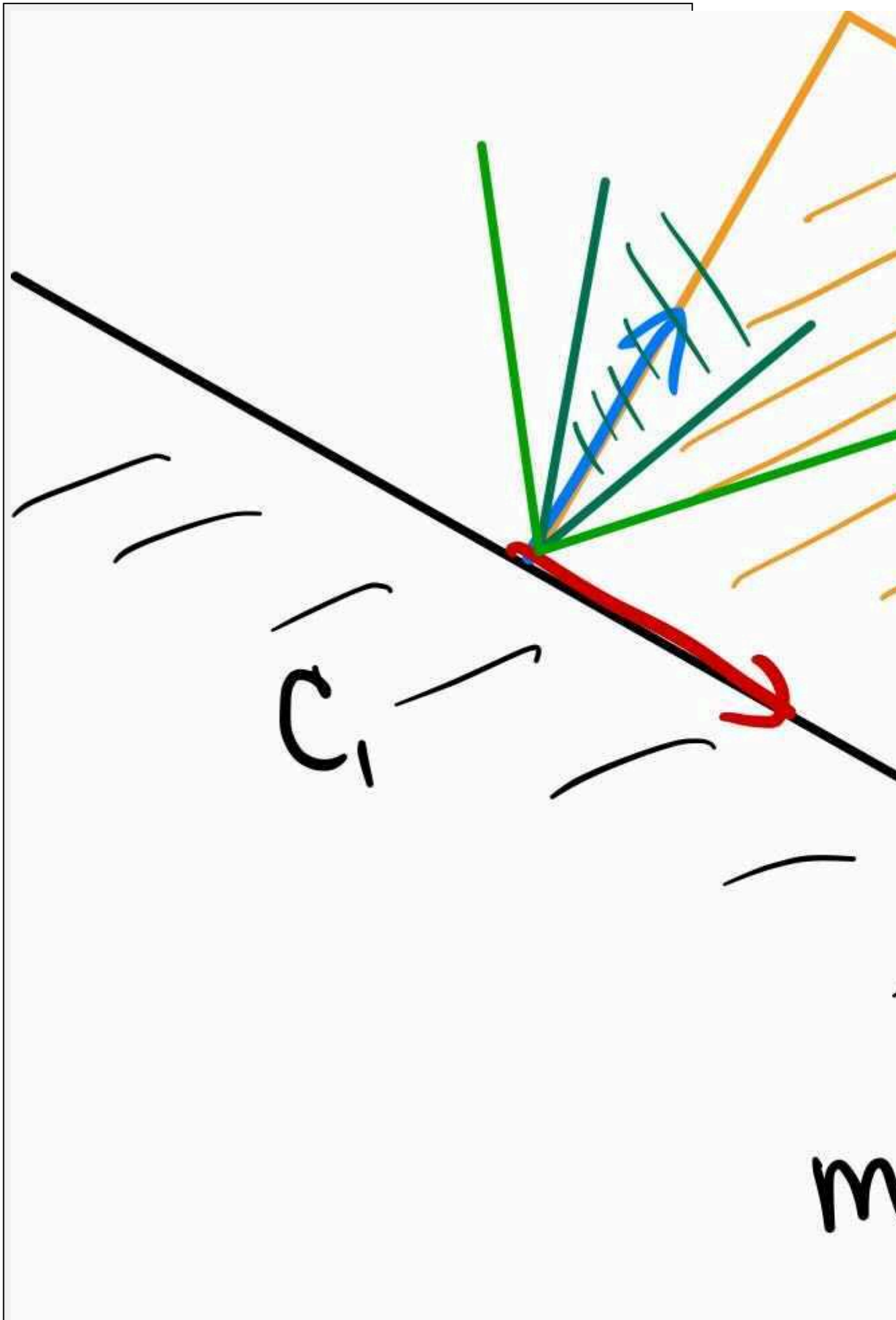
Taken together, the geometry of these constraints forms a cone of admissable contact forces. It is famously known as the "friction cone", and we will refer to it often.

It's a bit strange for us to write that the forces are in some set. Surely the world will pick just one force to apply? It can't apply all forces in a set. The friction cone specifies the range of possible forces; under the Coulomb friction model we say that the one force that will be picked is the force inside this set that will successfully resist relative motion in the contact x-y frame. If no force inside the friction cone can completely resist the motion, then we have sliding, and we say that the force that

the world will apply is the force inside the friction cone of *maximal dissipation*. For the conic friction cone, this will be pointed in the direction opposite of the sliding velocity. So even though the world will indeed "pick" one force from the friction cone to apply, it can still be meaningful to reason about the set of possible forces that could be applied because those denote the set of possible opposing forces that friction can perfectly resist. For instance, a brick under gravity will not move if we can exactly oppose the force of gravity with a force inside the friction cone.

Example 5.5 (Brick on an inclined half-plane)

If we take our last example, but tilt the table to an angle relative to gravity, then the horizontal forces start becoming important. Before going through the equations, let's check your intuition. Will the magnitude of the forces on the two corners stay the same in this configuration? Or will there be more contact force on the lower corner?


c,

In the illustration above, you'll notice that the contact frames have rotated so that the z axis is aligned with the contact normals. I've sketched two possible friction cones (dark green and lighter green), corresponding to two different coefficients of friction. We can tell immediately by inspection that the smaller value of μ (corresponding to the darker green) cannot produce contact forces that will completely counteract gravity (the friction cone does not contain the vertical line). In this case, the box will slide and no static equilibrium exists in this configuration.

If we increase the coefficient of (static) friction to the range corresponding to the lighter green, then we can find contact forces that produce an equilibrium. Indeed, for this problem, we need some amount of friction to even have an equilibrium (we'll explore this in the exercises). We also need for the vertical projection of the center of mass onto the ramp to land between the two contact points, otherwise the brick will tumble over the bottom edge. We can see this by writing our same force/torque balance equations. We can write them in body frame, B, assuming the center of mass in the center of the brick and the brick has length l and height h:

$$\begin{aligned} f_{B_x}^{c_1} + f_{B_x}^{c_2} &= -mg\sin\gamma \\ f_{B_z}^{c_1} + f_{B_z}^{c_2} &= mg\cos\gamma \\ -hf_{B_x}^{c_1} + lf_{B_z}^{c_1} &= hf_{B_x}^{c_2} + lf_{B_z}^{c_2} \\ f_{B_z}^{c_1} \geq 0, &\quad f_{B_z}^{c_2} \geq 0 \\ |f_{B_x}^{c_1}| \leq \mu f_{B_z}^{c_1}, &\quad |f_{B_x}^{c_2}| \leq \mu f_{B_z}^{c_2} \end{aligned}$$

So, are the magnitude of the contact forces the same or different? Substituting the first equation into the third reveals

$$f_{B_z}^{c_2} = f_{B_z}^{c_1} + \frac{mgh}{l}\sin\gamma.$$

Static equilibrium as an optimization problem

Rather than dropping objects from a random height, perhaps we can initialize our simulations using optimization to find the initial conditions that are already in static equilibrium. In DRAKE, the `StaticEquilbriumProblem` collects all of the constraints we enumerated above into an optimization problem:

$$\begin{aligned} \text{find}_q \quad \text{subject to} \quad & \tau_g(q) = -\sum_i J_i^T(q) f^{c_i} \\ & f_{C_z}^{c_i} \geq 0 && \forall i, \\ & |f_{C_{x,y}}^{c_i}|_2 \leq \mu f_{C_z}^{c_i} && \forall i, \\ & \phi_i(q) \geq 0 && \forall i, \\ & \phi(q) = 0 \textbf{ or } f_{C_z}^{c_i} = 0 && \forall i, \\ & \text{joint limits.} \end{aligned}$$

This is a nonlinear optimization problem: it includes the nonconvex non-penetration constraints we discussed in the last chapter. The second-to-last constraints (a logical or) is particularly interesting; constraints of the form $x \geq 0, y \geq 0, x = 0$ **or** $y = 0$ are known as complementarity constraints, and are often written as $x \geq 0, y \geq 0, xy = 0$. We can make the problem easier for the nonlinear optimization solver by relaxing the equality to $0 \leq \phi(q) f_{C_z}^{c_i} \leq \text{tol}$, which provides a proper gradient for the optimizer to follow at the cost of allowing some force at a distance.

It's easy to add additional costs and constraints; for initial conditions we might use an objective that keeps q close to an initial configuration-space sample.

Example 5.6 (Tall Towers)

So how well does it work?

5.2 A FEW OF THE NUANCES OF SIMULATING CONTACT

Example 5.7 (Contact force inspector)

I've created a simple GUI that allows you to pick any two primitive geometry types and inspect the contact information that is computed when those object are put into penetration. There is a lot of information displayed there! Take a minute to make sure you understand the colors, and the information provided in the textbox display.

When I play with the GUI above, I feel almost overwhelmed. First, overwhelmed by the sheer number of cases that we have to get right in the code; it's unfortunately extremely common for open-source tools to have bugs in here. But second, overwhelmed by the sense that we are asking the wrong question. Asking to summarize the forces between two bodies in deep penetration with a single Cartesian force applied at a point is fraught with peril. As you move the objects, you will find many discontinuities; this is a common reason why you sometimes see rigid body simulations "explode". It might seem natural to try to use multiple contact points instead of a single contact point to summarize the forces, and some simulators do, but it is very hard to write an algorithm that only depends on the current configurations of the geometry which applies forces consistently from one time step to the next with these approaches.

I currently feel that the only fundamental way to get around these numerical issues is to start reasoning about the entire volume of penetration. The Drake developers have recently proposed a version of this that we believe we can make computationally tractable enough to be viable for real-time simulation [3], and are working hard to bring this into full service in Drake. You will find many references to "hydroelastic" contact throughout the code. We hope to turn it on by default soon.

In the mean time, we can make simulations robust by carefully curating the contact geometry...

5.3 MODEL-BASED GRASP SELECTION

What makes a good grasp? This topic has been studied extensively for decades in robotics, with an original focus on thinking of a (potentially dexterous) hand interacting with a known object. [4] is an excellent survey of that literature; I will summarize just a few of the key ideas here. To do it, let's start by extended our spatial algebra to include forces.

5.3.1 Spatial force

In our discussion of contact forces above, we started by thinking of a force as a three-element vector (with components for x, y, and z) applied to a rigid body at some point. More generally, we will define a six-component vector for *spatial force*:

$$F^{B_p}_{\text{name},C} = \begin{bmatrix} \tau^{B_p}_{\text{name},C} \\ f^{B_p}_{\text{name},C} \end{bmatrix} \quad \text{or} \quad \left[F^{B_p}_{\text{name}}\right]_C = \begin{bmatrix} \left[\tau^{B_p}_{\text{name}}\right]_C \\ \left[f^{B_p}_{\text{name}}\right]_C \end{bmatrix}. \tag{1}$$

$F^{B_p}_{\text{name},C}$ is the named spatial force applied to a point, or frame origin, B_p, expressed in frame C. The form with the parentheses is preferred in Drake, but is a verbose for my taste here in the notes. The name is optional, and the expressed in frame, if unspecified, is the world frame. For forces in particular, it is recommended that we include the body, B, that the force is being applied to in the symbol for the point B_p, especially since we will often have equal and opposite forces. In code, we write

`Fname_Bp_C`.

Like spatial velocity, spatial forces have a rotational component and a translational component; $\tau_C^{B_p} \in \mathbb{R}^3$ is the *torque* (on body B applied at point p expressed in frame C), and $f_C^{B_p} \in \mathbb{R}^3$ is the translational or Cartesian force. A spatial force is also commonly referred to as a wrench. If you find it strange to think of forces as having a rotational component, then think of it this way: the world might only impart Cartesian forces at the points of contact, but we want to summarize the combined effect of many Cartesian forces applied to different points into one common frame. To do this, we represent the equivalent effect of each Cartesian force at the point of application as a force + torque applied at a different point on the body.

Spatial forces fit neatly into our spatial algebra:

- Spatial forces add when they are applied to the same body in the same frame, e.g.:

$$F_{\text{total},C}^{B_p} = \sum_i F_{i,C}^{B_p}. \tag{2}$$

- Shifting a spatial force from one application point, B_p, to another point, B_q, uses the cross product:

$$f_C^{B_q} = f_C^{B_p}, \qquad \tau_C^{B_q} = \tau_C^{B_p} + {}^{B_q}p_C^{B_p} \times f_C^{B_p}. \tag{3}$$

- As with all spatial vectors, rotations can be used to change between the "expressed-in" frames:

$$f_D^{B_p} = {}^D R^C f_C^{B_p}, \qquad \tau_D^{B_p} = {}^D R^C \tau_C^{B_p}. \tag{4}$$

Now we are starting to have the vocabulary to provide one answer to the question: what makes a good grasp? If the goal of a grasp is to stabilize an object in the hand, then a good grasp can be one that is able to resist disturbances described as an "adversarial" wrench applied to the body.

5.3.2 The contact wrench cone

Above, we introduced the friction cone as the range of possible forces that friction is able to produce in order to resist motion. For the purposes of grasp planning, by applying the additive inverse to the forces in the friction cone, we can obtain all of the "adversarial" forces that can be resisted at the point of contact. And to understand the total ability of all contact forces (applied at multiple points of contact) to resist motion of the body, we want to somehow add all of these forces together. Fortunately, the spatial algebra for spatial forces can be readily extended from operating on spatial force vectors to operating on entire sets of spatial forces.

Because our sets of interest here are convex cones, I will use the relatively standard choice of $\mathcal{K}$ for the six-dimensional *wrench cone*. Specifically, we have $\mathcal{K}_{\text{name},C}^{B_p}$ for the cone corresponding to potential spatial forces for $F_{\text{name},C}^{B_p}$. For instance, our Coulomb friction cone for the point contact model (which, as we've defined it, has no torsional friction) in the contact frame could be:

$$\mathcal{K}_C^C = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right. \left. : \sqrt{\left(f_{C_x}^C\right)^2 + \left(f_{C_y}^C\right)^2} \le \mu f_{C_z}^C \right\}. \tag{5}$$

The spatial algebra for spatial forces can be applied directly to the wrench cones:

- For addition of wrenches applied at the same point and expressed in the same frame, the interpretation we seek is that the cone formed by the sum of wrench cones describes the set of wrenches that could be obtained by choosing one element from each of the individual cones and summing them up. This set operator is the Minkowski sum, which we denote with $\oplus$, e.g.:

$$\mathcal{K}^{B_p}_{\text{total},C} = \mathcal{K}^{B_p}_{0,C} \oplus \mathcal{K}^{B_p}_{1,C} \oplus \cdots \tag{6}$$

- Shifting a wrench cone from one application frame, B_p, to another frame, B_q, is a linear operation on the cones; to emphasize that I will write Eq 3 in matrix form:

$$\mathcal{K}^{B_q}_{C} = \begin{bmatrix} I_{3\times3} & \left[{}^{B_q}p^{B_p}_{C}\right]_\times \\ 0_{3\times3} & I_{3\times3} \end{bmatrix} \mathcal{K}^{B_p}_{C}, \tag{7}$$

  where the notation $[p]_\times$ is the skew-symmetric matrix corresponding to the cross product.
- Rotations can be used to change between the "expressed-in" frames:

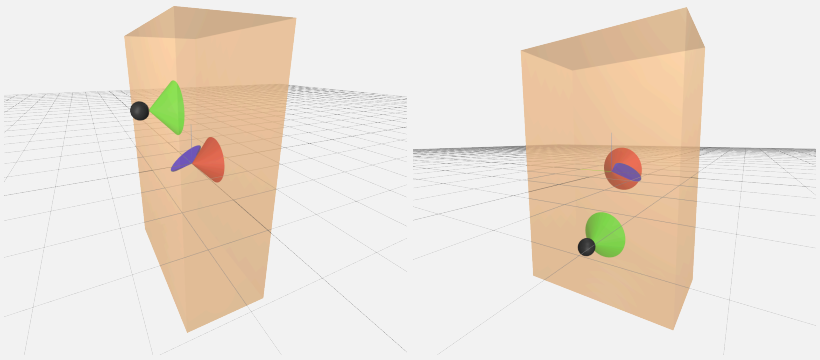
$$\mathcal{K}^{B_p}_{D} = \begin{bmatrix} {}^{D}R^{C} & 0_{3\times3} \\ 0_{3\times3} & {}^{D}R^{C} \end{bmatrix} \mathcal{K}^{B_p}_{C}. \tag{8}$$

Example 5.8 (A contact wrench cone visualization)

I've made a simple interactive visualization for you to play with to help your intuition about these wrench cones. I have a box that fixed in space and only subjected to contact forces (no gravity is illustrated here); I've drawn the friction cone at the point of contact and at the center of the body.

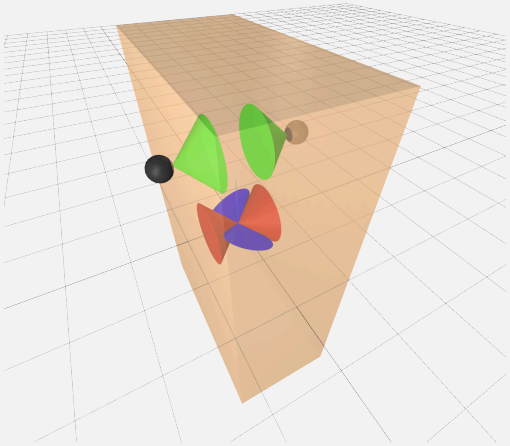
There is one major caveat: the wrench cone lines in $\mathbb{R}^6$, but I can only draw cones in $\mathbb{R}^3$. So I've made the (standard) choice to draw the projection of the 6d cone into 3d space with two cones: one for the translational components (in green for the contact point and again in red for the body frame) and another for the rotational components (in blue for the body frame). This can be slightly misleading because one cannot actually select independently from both sets.

Here is the contact wrench cone for a single contact point, visualized for two different contact locations:



I hope that you immediately notice that the rotational component of the wrench cone is low dimensional -- due to the cross product, all vectors in that space must be orthogonal to the vector ${}^{B}p^{C}$. Of course it's way better to run the notebook yourself and get the 3d interactive visualization.

Here is the contact wrench cone for a two contact points on that are directly opposite from each other:



Notice that while each of the rotational cones are low dimensional, they span different spaces. Together (as understood by the Minkowski sum of these two sets) they can resist all pure torques applied at the body frame. This intuition is largely correct, but this is also where the projection of the 6d cone onto two 3d cones becomes a bit misleading.there are some wrenches that cannot be resisted by this grasp. Specifically, if I were to visualize the wrench cone at the point directly between the two contact points, we would see that the wrench cones *do not include* torques applied directly along the axis between the two contacts. The two contact points alone, without torsional friction, are unable to resist torques about that axis.

In practice, the gripper model that we use in our simulation workflow has multiple contact points places along the round surface of the gripper; even though each of them alone has no torsional friction the net wrench from these multiple points of contact provides some amount of wrench in the axis between the fingers. The exact amount one gets will be proportional to how hard the gripper is squeezing the object.

Now we can compute the cone of possible wrenches that any set of contacts can apply on a body -- the contact wrench cone -- by putting all of the individual contact wrench cones into a common frame and summing them together. A classic metric for grasping would say that a good grasp is one where the contact wrench cone is large (can resist many adversarial wrench disturbances). If the contact wrench cone is all of $\mathbb{R}^6$, then we say the contacts have a achieved *force closure*[4].

It's worth mentioning that the elegant (linear) spatial algebra of the wrench cones also makes these quantities very suitable for use in optimization (e.g. [5]).

5.3.3 Colinear antipodal grasps

The beauty of this wrench analysis originally inspires a very model-based analysis of grasping, where one could try to optimize the contact locations in order to maximize the contact wrench cone. But our goals for this chapter are to assume very little about the object that we are grasping, so we'll (for now) avoid optimizing over the surface of an object model for the best grasp location. Nevertheless, our model-based grasp analysis gives us a few very good heuristics for grasping even unknown objects.

In particular, a good heuristic for a two fingered gripper to have a large contact wrench cone is to find colinear "antipodal" grasp points. Antipodal here means that

the normal vectors of the contact (the z axis of the contact frames) are pointing in exactly opposite directions. And "colinear" means that they are on the same line -- the line between the two contact points. As you can see in the two-fingered contact wrench visualization above, this is a reasonably strong heuristic for having a large total contact wrench cone. As we will see next, we can apply this heuristic even without knowing much of anything about the objects.

5.4 Grasp selection from point clouds

Rather than looking into the bin of objects, trying to recognize specific objects and estimate their pose, the newer approach to grasp selection that has proven incredibly useful for bin picking is to just look for graspable areas directly on the (unsegmented) point cloud. Some of the most visible proponents of this approach are [6, 7, 8]. If you look at those references, you'll see that they all use learning to somehow decide where in the point cloud to grasp. And I **do** think learning has a lot to offer in terms of choosing good grasps -- it's a nice and natural learning formulation and there is significant information besides just what appears in the immediate sensor returns that can contribute to deciding where to grasp. But often you will see that underlying those learning approaches is an approach to selecting grasps based on geometry alone, if only to produce the original training data. I also think that the community doesn't regularly acknowledge just how well the purely geometric approaches still work. I'd like to discuss those first.

5.4.1 Point cloud pre-processing

To get a good view into the bin, we're going to set up multiple RGB-D cameras. I've used three per bin in all of my examples here. And those cameras don't only see the objects in the bin; they also see the bins, the other cameras, and anything else in the viewable workspace. So we have a little work to do to merge the point clouds from multiple cameras into a single point cloud that only includes the area of interest.

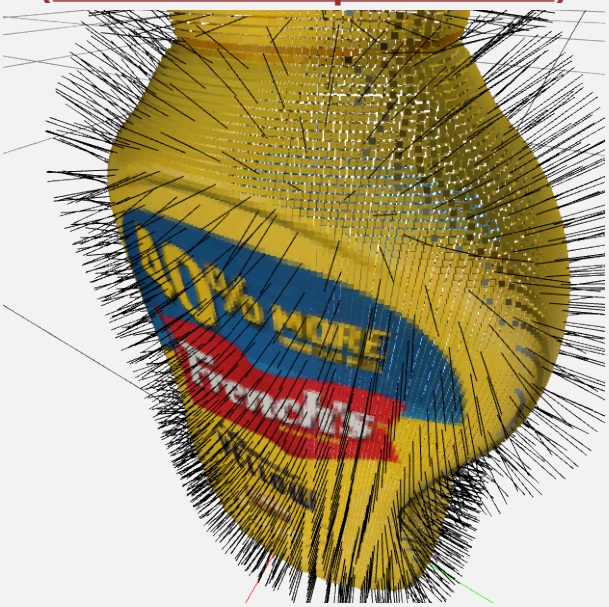
First, we can *crop* the point cloud to discard any points that are from outside the area of interest (which we'll define as an axis-aligned bounding box immediately above the known location of the bin).

As we will discuss in some detail below, many of our grasp selection strategies will benefit from *estimating the "normals"* of the point cloud (a unit vector that estimates the normal direction relative to the surface of the underlying geometry). It is actually better to estimate the normals on the individual point clouds, making use of the camera location that generated those points, than to estimate the normal after the point cloud has been merged.

For sensors mounted on the real world, *merging point clouds* requires high-quality camera calibration and must deal with the messy depth returns. All of the tools from the last chapter are relevant, as the tasks of merging the point clouds is another instance of the point-cloud-registration problem. For the perfect depth measurements we can get out of simulation, given known camera locations, we can skip this step and simply concatenate the list of points in the point clouds together.

Finally, the resulting raw point clouds likely include many more points then we actually need for our grasp planning. One of the standard approaches for *down-sampling* the point clouds is using a voxel grid -- regularly sized cubes tiled in 3D. We then summarize the original point cloud with exactly one point per voxel (see, for instance Open3D's note on voxelization). Since point clouds typically only occupy a small percentage of the voxels in 3D space, we use sparse data structures to store the voxel grid. In noisy point clouds, this voxelization step is also a useful form of filtering.

Example 5.9 (Mustard bottle point clouds)



I've produced a scene with three cameras looking at our favorite YCB mustard bottle. I've taken the individual point clouds (already converted to the world frame by the `DepthImageToPointCloud` system), cropped the point clouds (to get rid of the geometry from the other cameras), estimated their normals, merged the point clouds, then down-sampled the point clouds. The order is important!

I've pushed all of the point clouds to meshcat, but with many of them set to not be visible by default. Use the drop-down menu to turn them on and off, and make sure you understand basically what is happening on each of the steps. For this one, I can also give you the meshcat output directly, if you don't want to run the code.

5.4.2 Estimating normals and local curvature

The grasp selection strategy that we will develop below will be based on the local geometry (normal direction and curvature) of the scene. Understanding how to estimate those quantities from point clouds is an excellent exercise in point cloud processing, and is representative of other similar point cloud algorithms.

Let's think about the problem of fitting a plane, in a least-squares sense, to a set of points[9]. We can describe a plane in 3D with a position p and a unit length normal vector, n. The distance between a point p^i and a plane is simply given by the magnitude of the inner product, $(p^i - p)^T n$. So our least-squares optimization becomes

$$\min_{p,n} \quad \sum_{i=1}^{N} (p^i - p)^T n \ ^2, \quad \text{subject to} \quad |n| = 1.$$

Taking the gradient of the Lagrangian with respect to p and setting it equal to zero gives us that

$$p^* = \frac{1}{N} \sum_{i=1}^{N} p^i.$$

Inserting this back into the objective, we can write the problem as

$$\min_n n^T W n, \quad \text{subject to} \quad |n| = 1, \quad \text{where } W = \left[\sum_i (p^i - p^*)(p^i - p^*)^T\right].$$

Geometrically, this objective is a quadratic bowl centered at the origin, with a unit circle constraint. So the optimal solution is given by the (unit-length) eigenvector corresponding to the *smallest* eigenvalue of the data matrix, W. And for any optimal n, the "flipped" normal $-n$ is also optimal. We can pick arbitrarily for now, and then flip the normals in a post-processing step (to make sure that the normals all point towards the camera origin).

What is really interesting is that the second and third eigenvalues/eigenvectors also tell us something about the local geometry. Because W is symmetric, it has orthogonal eigenvectors, and these eigenvectors form a (local) basis for the point cloud. The smallest eigenvalue pointed along the normal, and the *largest* eigenvalue corresponds to the direction of least curvature (the squared dot product with this vector is the largest). This information can be very useful for finding and planning grasps. [6] and others before them use this as a primary heuristic in generating candidate grasps.

In order to approximate the local curvature of a mesh represented by a point cloud, we can use our fast nearest neighbor queries to find a handful of local points, and use this plane fitting algorithm on just those points. When doing normal estimation directly on a depth image, people often forgo the nearest-neighbor query entirely; simply using the approximation that neighboring points in pixel coordinates are often nearest neighbors in the point cloud. We can repeat that entire procedure for every point in the point cloud.

I remember when working with point clouds started to become a bigger part of my life, I thought that surely doing anything moderately computational like this on every point in some dense point cloud would be incompatible with online perception. But I was wrong! Even years ago, operations like this one were often used inside real-time perception loops. (And they pale in comparison to the number of FLOPs we spend these days evaluating large neural networks).

Example 5.10 (Normals and local curvature of the mustard bottle.)

I've coded up the basic least-squares surface estimation algorithm, with the query point in green, the nearest neighbors in blue, and the local least squares estimation drawn with our RGB⇔XYZ frame graphic. You should definitely slide it around and see if you can understand how the axes line up with the normal and local curvature.

You might wonder where you can read more about algorithms of this type. I don't have a great reference for you. But Radu Rusu was the main author of the point cloud library[10], and his thesis has a lot of nice summaries of the point cloud algorithms of 2010[11].

5.4.3 Evaluating a candidate grasp

Now that we have processed our point cloud, we have everything we need to start planning grasps. I'm going to break that discussion down into two steps. In this section we'll come up with a cost function that scores grasp candidates. In the following section, we'll discuss some very simple ideas for trying to find grasps candidates that have a low cost.

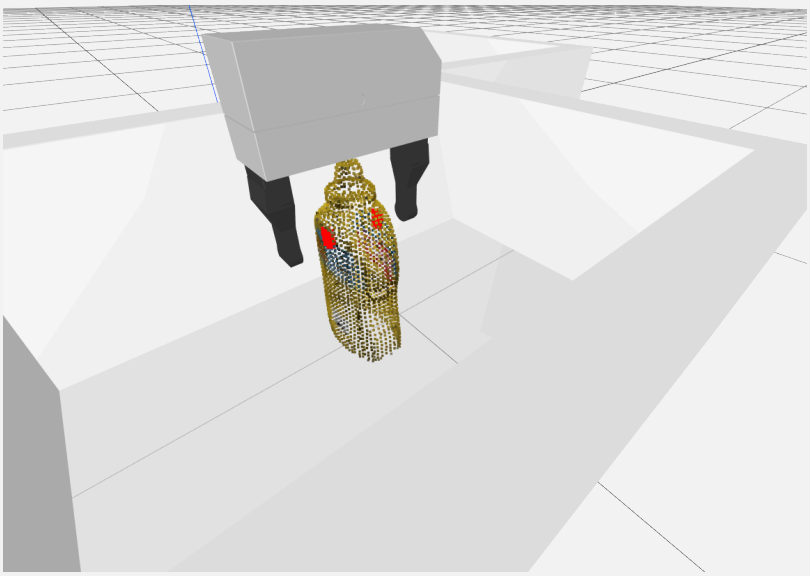
Following our discussion of "model-based" grasp selection above, once we pick up an object -- or whatever happens to be between our fingers when we squeeze -- then we will expect the contact forces between our fingers to have to resist at least the *gravitational wrench* (the spatial force due to gravity) of the object. The closing force provided by our gripper is in the gripper's x-axis, but if we want to be able to pick up the object without it slipping from our hands, then we need forces inside the friction cones of our contacts to be able to resist the gravitational wrench. Since we don't know what that wrench will be (and are somewhat constrained by the geometry of our fingers), a reasonable strategy is to look the colinear antipodal points on the surface of the point cloud which also align with x-axis of the gripper. In a real point cloud, we are unlikely to find perfect antipodal pairs, but finding areas with normals pointing in nearly opposite directions is a good strategy for grasping!

Example 5.11 (Scoring grasp candidates)

In practice, the contact between our fingers and the object(s) will be better described by a *patch contact* than by a point contact (due to the deflection of the rubber fingertips and any deflection of the objects being picked). So it makes sense to look for patches of points with agreeable normals. There are many ways one could write this, I've done it here by transforming the processed point cloud of the scene into the candidate frame of the gripper, and cropped away all of the points except the ones that are inside a bounding box between the finger tips (I've marked them in red in MeshCat). The first term in my grasping cost function is just reward for all of the points in the point cloud, based on how aligned their normal is to the x-axis of the gripper:

$$\text{cost} = -\sum_i (n^i_{G_x})^2,$$

where $n^i_{G_x}$ is the x component of the ith point in the cropped point cloud expressed in the gripper frame.



There are other considerations for what might make a good grasp, too. For our kinematically limited robot reaching into a bin, we might favor grasps that put the hand in favorable orientation for the arm. In the grasp metric I've implemented in the code, I added a cost for the hand deviating from vertical. I can reward the dot product of the vector world $-z$ vector, $[0, 0, -1]$ with the y-axis in gripper frame rotated into world frame with :

$$\text{cost } += -\alpha \begin{bmatrix} 0 & 0 & -1 \end{bmatrix} R^G \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \alpha R^G_{3,2},$$

where α is relative cost weight, and $R^G_{3,2}$ is the scalar element of the rotation matrix in the third row and second column.

Finally, we need to consider collisions between the candidate grasp and both the bins and with the point cloud. I simply return infinite cost when the gripper is in collision. I've implemented all of those terms in the notebook, and given you a sliders to move the hand around and see how the cost changes.

5.4.4 Generating grasp candidates

We've defined a cost function that, given a point cloud from the scene and a model of the environment (e.g. the location of the bins), can score a candidate grasp pose, X^G. So now we would like to solve the optimization problem: find X^G that minimizes the cost subject to the collision constraints.

Unfortunately, this is an extremely difficult optimization problem, with a highly nonconvex objective and constraints. Moreover, the cost terms corresponding to the antipodal points is zero for most X^G -- since most random X^G will not have any points between the fingers. As a result, instead of using the typical mathematical programming solvers, most approaches in the literature resort to a randomized sampling-based algorithm. And we do have strong heuristics for picking reasonable samples.

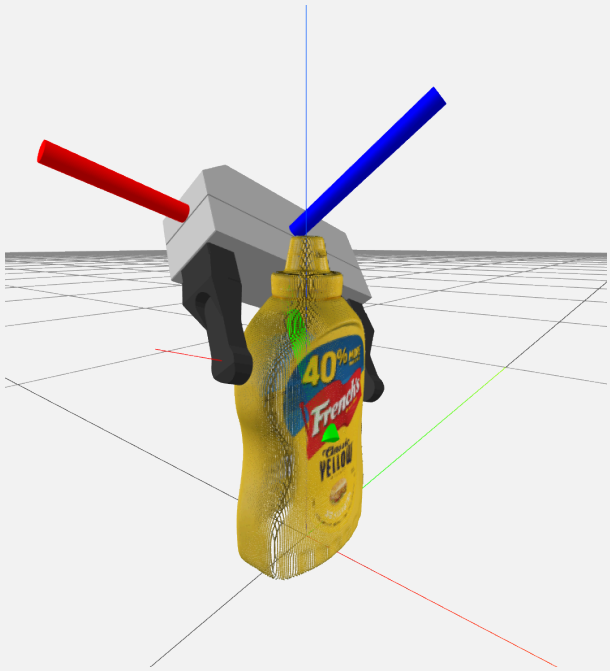
One heuristic, used for instance in [6], is to use the local curvature of the point cloud to propose grasp candidates that have the point cloud normals pointing into the palm, and orients the hand so that the fingers are aligned with the direction of maximum curvature. One can move the hand in the direction of the normal until the fingers are out of collision, and even sample nearby points. We have written an

exercise for you to explore this heuristic. But for our YCB objects, I'm not sure it's the best heuristic; we have a lot of boxes, and boxes don't have a lot of information to give in their local curvature.

Another heuristic is to find antipodal point pairs in the point cloud, and then sample grasp candidates that would align the fingers with those antipodal pairs. Many 3D geometry libraries support "ray casting" operations at least for a voxel representation of a point cloud; so a reasonable approach to finding antipodal pairs is to simply choose a point at random in the point cloud, then ray cast into the point cloud in the opposite direction of the normal. If the normal of the voxel found by ray casting is sufficiently antipodal, and if the distance to that voxel is smaller than the gripper width, then we've found a reasonable antipodal point pair.

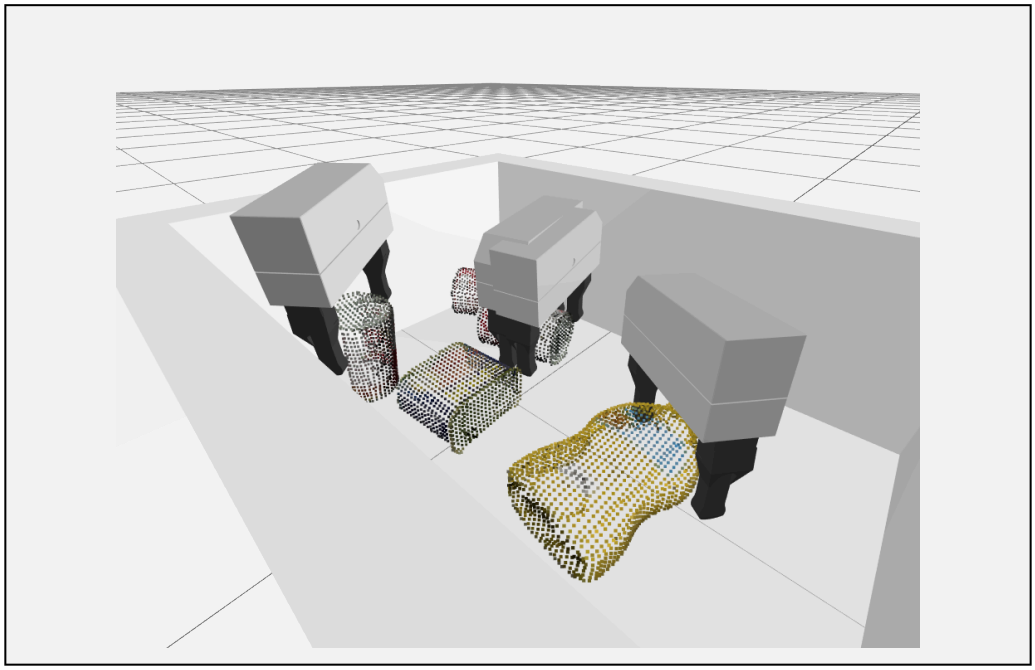
Example 5.12 (Generating grasp candidates)

As an alternative to ray casting, I've implemented an even simpler heuristic in my code example: I simply choose a point at random, and start sampling grasps in orientation (starting from vertical) that align the x-axis of the gripper with the normal of that point. Then I mostly just rely on the antipodal term in my scoring function to allow me to find good grasps.



I do implement one more heuristic -- once I've found the points in the point cloud that are between the finger tips, then I move the hand out along the gripper x -axis so that those points are centered in the gripper frame. This helps prevent us knocking over objects as we close our hands to grasp them.

But that's it! It's a very simple strategy. I sample a handful of candidate grasps and just draw the top few with the lowest cost. If you run it a bunch of times, I think you will find it's actually quite reasonable. Every time it runs, it is simulating the objects falling from the sky; the actual grasp evaluation is actually quite fast.



5.5 The corner cases

If you play around with the grasp scoring I've implemented above a little bit, you will find deficiencies. Some of them are addressed easily (albeit heuristically) by adding a few more terms to the cost. For instance, I didn't check collisions of the pre-grasp configuration, but this could be added easily.

There are other cases where grasping alone is not sufficient as a strategy. Imagine that you place an object right in one of the corners of the bin. It might not be possible to get the hand around both sides of the object without being in collision with either the object or the side. The strategy above will never choose to try to grab the very corner of a box (because it always tried to align the sample point normal with the gripper x), and it's not clear that it should. This is probably especially true for our relatively large gripper. In the setup we used at TRI, we implemented an additional simple "pushing" heuristic that would be used if there were point clouds in the sink, but no viable grasp candidate could be found. Instead of grasping, we would drive the hand down and nudge that part of the point cloud towards the middle of the bin. This can actually help a lot!

There are other deficiencies to our simple approach that would be very hard to address with a purely geometric approach. Most of them come down to the fact that our system so far has no notion of "objects". For instance, it's not uncommon to see this strategy result in "double picks" if two objects are close together in the bin. For heavy objects, it might be important to pick up the object close to the center of mass, to improve our chances of resisting the gravitational wrench while staying in our friction cones. But our strategy here might pick up a heavy hammer by squeezing just the very far end of the handle.

Interestingly, I don't think the problem is necessarily that the point cloud information is insufficient, even without the color information. I could show you similar point clouds and you wouldn't make the same mistake. These are the types of examples where learning a grasp metric could potentially help. We don't need to achieve artificial general intelligence to solve this one; just experience knowing that when we tried to grasp in someplace before we failed would be enough to improve our grasp heuristics significantly.

5.6 Programming the Task Level

Most of us are used to writing code in a procedural programming paradigm; code executes from the top of the program through function calls, branchs (like if/then statements), for/while loops, etc. It's tempting to to to write our high-level robot programs like that, too. For the clutter task, I could write "while there is an object to pick up in the first bin, pick it up and move it to the second bin; ...". And that is certainly not wrong! But to be a component in a robotics framework, this high-level program must somehow communicate with the rest of the system at runtime, including having interfaces to the low-level controllers that are running at some regular frequency (say 100Hz or even 1kHz). So how do we author the high-level behaviors (often called "task level") in a way that integrates seamlessly with the high-rate low-level controls?

There are a few important programming paradigms here. Broadly speaking, I would categorize them into three bins:

- Procedural code (in a separate thread) passes messages/events to a module that lives in the main robot/simulation thread.
- Task-level "policies" can be authored directly as e.g. finite state machines (FSMs) or "behavior trees", which get evaluated directly in the robot/simulation loop.
- Task-level "planners" can take rules for how task-level behaviors can be chained together to achieve long-term goals:
    - In the simplest case, the planner outputs a plan that can be executed during the robot/simulation loop, just like the trajectories we've used to plan the motion of the gripper.
    - If the planner can be run online (e.g. constantly adjusting/replanning given new sensor information), then it can be used directly in the robot/simulation loop.
    - Alternatively, a few planners can actually "compile" their plans directly into a policy (e.g. in the form of an FSM). Here is one of the classic examples.

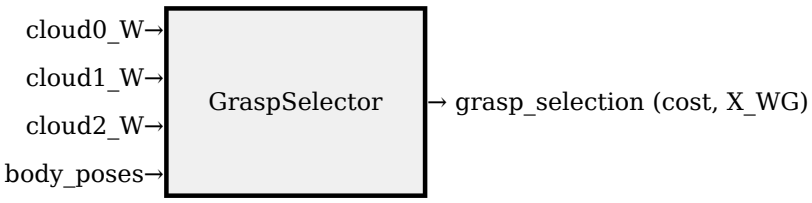
Each of these paradigms has its place, and some researchers heavily favor one over the other. I'll always remember Rod Brooks writing papers like Elephants Don't Play Chess to argue against task-level planning[12, 13, 14]. Rod's "subsumption architecture" was distinctly in the "task-level policies" category and can be seen as a precursor to Behavior Trees; it established its credibility by having success on the early Roomba vacuum cleaners. The planning community might counter that for truly complicated tasks, it would be impossible for human programmers to author a policy for all possible permutations / situations that an AI system might encounter; that planning is the approach that truly scales. They have addressed some of the initial criticisms about symbol grounding and modeling requirements with extensions like planning in belief space and task-and-motion planning, which we will cover in later parts of the notes.

For the purposes of clutter clearing in the systems framework, taking the second approach: writing the task-level behavior "policy" as a simple state machine will work nicely.

Example 5.13 (A simple state machine for "clutter clearing")

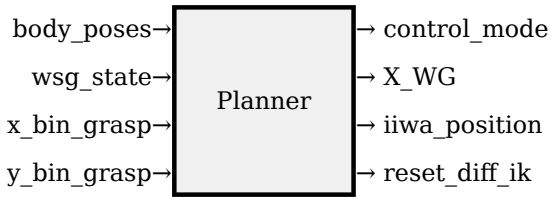
5.7 Putting it all together

First, we bundle up our grasp-selection algorithm into a system that reads the point clouds from 3 cameras, does the point cloud processing (including normal estimation), and the random sampling, and puts the grasp selection onto the output port:



Note that we will instantiate this system twice -- once for the bin located on the positive X axis and again for the system on the negative y axis. Thanks to the "pull-architecture" in Drake's systems framework, this relatively expensive sampling procedure only gets called when the downstream system evaluates the output port.

The work-horse system in this example is the `Planner` system which implements the basic state machine, calls the planning functions, stores the resulting plans in its `Context`, and evaluates the plans on the output port:



This system needs enough input / output ports to get the relevant information into the planner, and to command the downstream controllers.

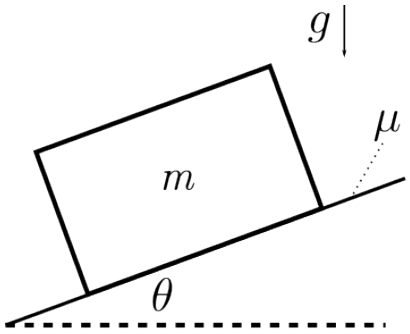
Example 5.14 (Clutter clearing (complete demo))

5.8 Exercises

Exercise 5.1 (Assessing static equilibrium)

For this problem, we'll use the Coulomb friction model, where $|f_t| \leq \mu f_n$. In other words, the friction force is large enough to resist any movement up until the force required would exceed μf_n, in which case $|f_t| = \mu f_n$.

a. Consider a box with mass m sitting on a ramp at angle θ, with coefficient of box μ in between the sphere and the ramp:



For a given angle θ, what is the minimum coefficient of friction required for the box to not slip down the plane? Use g for acceleration due to

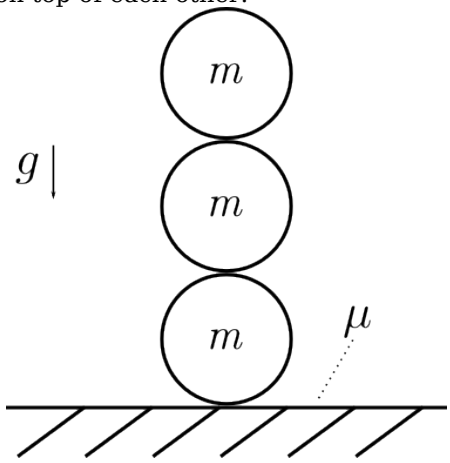
gravity.

Now consider a flat ground plane with three solid (uniform density) spheres sitting on it, with radius r and mass m. Assume they have the same coefficient of friction μ between each other as with the ground.

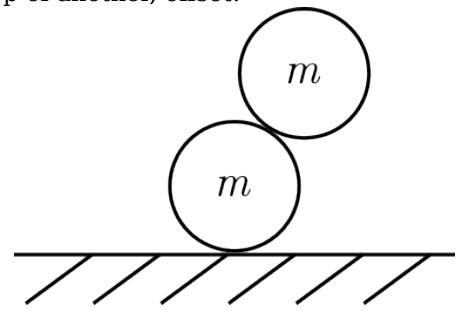
For each of the following configurations: could the spheres be in static equilibrium for some $\mu \in [0, 1]$, $m > 0$, $r > 0$? Explain why or why not. Remember that both torques and forces need to be balanced for all bodies to be in equilibrium.

To help you solve these problems, we have to help you build intuition and test your answers. It lets you specify the configuration of the spheres and then uses the StaticEquilbriumProblem class to solve for static equilibrium. Use this notebook to help visualize and think about the system, but for each of the configurations, you should have a physical explanation for your answer. (An example of such a physical explanation would be a free body diagram of the forces and torques on each sphere, and equations for how they can or cannot sum to zero. This is essentially what StaticEquilbriumProblem checks for.)

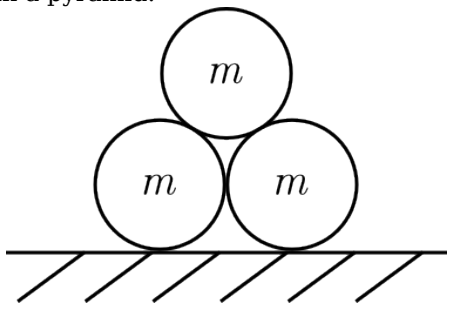
b. Spheres stacked on top of each other:



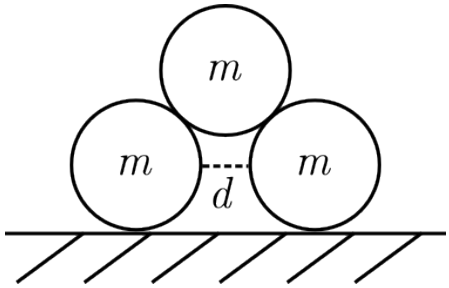
c. One sphere on top of another, offset:



d. Spheres stacked in a pyramid:



e. Spheres stacked in a pyramid, but with a distance d in between the bottom two:



Finally, a few conceptual questions on the StaticEquilbriumProblem:

f. Why does it matter what initial guess we specify for the system? (Hint: what type of optimization problem is this?)
g. Take a look at the Drake documentation for StaticEquilbriumProblem. It lists the constraints that are used when it's solve for equilibrium. Which two of these can a free body diagram answer?

Exercise 5.2 (Normal Estimation from Depth)

For this exercise, you will investigate a slightly different approach to normal vector estimation. In particular, we can exploit the spatial structure that is already in a depth image to avoid computing nearest neighbors. You will work exclusively in . You will be asked to complete the following steps:

a. Implement a method to estimate normal vectors from a depth image, without computing nearest neighbors.
b. Reason about a scenario where the depth image-based solution will not be as performant as computing nearest-neighbors.

Exercise 5.3 (Analytic Antipodal Grasping)

So far, we have used sampling-based methods to find antipodal grasps - but can we have find one analytically if we knew the equation of the object shape? For this exercise, you will analyze and implement a correct-by-construction method to find antipodal points using symbolic differentiation and MathematicalProgram. You will work exclusively in . You will be asked to complete the following steps:

a. Prove that antipodal points are critical points of an energy function defined on the shape.
b. Prove the converse does not hold.
c. Implement a method to find these antipodal points using MathematicalProgram.
d. Analyze the Hessian of the energy function and its relation to the type of antipodal grasps.

Exercise 5.4 (Grasp Candidate Generation)

In the chapter Colab notebook, we generated grasp candidates using the antipodal heuristic. In this exercise, we will investigate an alternative method for generating grasp candidates based on local curvature, similar to the one proposed in [6]. This exercise is implementation-heavy, and you will work exclusively in .

Exercise 5.5 (Behavior Trees)

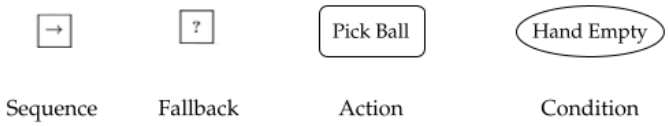
Let's reintroduce the terminology of behavior trees. Behavior trees provide a structure for switching between different tasks in a way that is both modular and reactive. Behavior trees are a directed rooted tree where the internal nodes manage the control flow and the leaf nodes are actions to be executed or conditions to be evaluated. For example, an action may be to "pick ball". This action can succeed or fail. A condition may be "Is my hand empty?" which can be true (thus the condition succeeds) or can be false (the condition fails).

We'll consider two categories of control flow nodes:

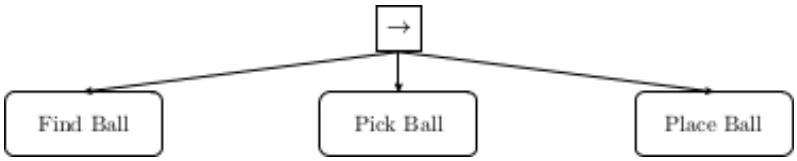
Sequence Node: (→)
Sequence nodes execute each of the child behaviors one after another. The sequence *fails if any of the children fail*. One way to think about this operator is that a sequence node takes an "and" over all of the child behaviors.

Fallback Node: (?)
Fallback nodes also execute each of the child behaviors one after another. However, fallback *succeeds if any of the children succeed*. One way to think about this operator is that the fallback node takes an "or" over all of the children behaviors.

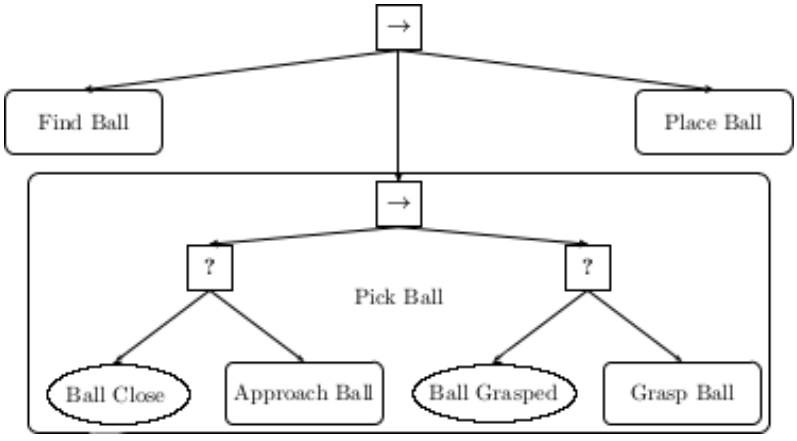
The symbols are visualized below. Sequence nodes are represented as an arrow in a box, fallback nodes are represented as a question mark in a box, actions are inside boxes and conditions are inside ovals.



Let's apply our understanding of behavior trees in the context of a simple task where a robot's goal is to: find a ball, pick the ball up and place the ball in a bin. We can describe task with the high-level behavior tree:



Confirm to yourself that this small behavior tree captures the task! We can go one level deeper and expand out our "Pick Ball" behavior:



The pick ball behavior can be considered as such: Let's start with the left branch. If the ball is already close, the condition "Ball Close?" returns true. If the ball is not close, we execute an action to "Approach ball", thus making the ball close. Thus either the ball is already close (i.e. the condition "Ball Close?" is true) or we approach the ball (i.e. the action "Approach ball" is successfully executed) to make the ball close. Given that the ball is now close, we consider the right branch of the pick action. If the ball is already grasped, the condition "Ball Grasped?" returns true. if the ball is not grasped, we execute the action, "Grasp Ball".
We can expand our our behavior tree even further to give our full behavior:

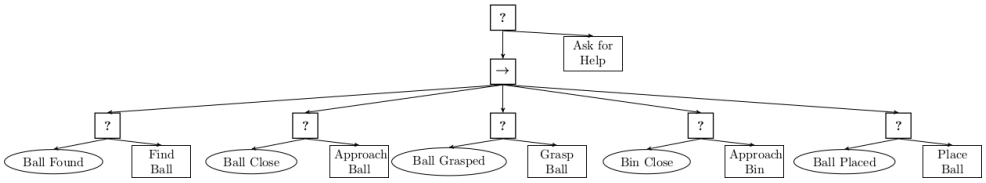
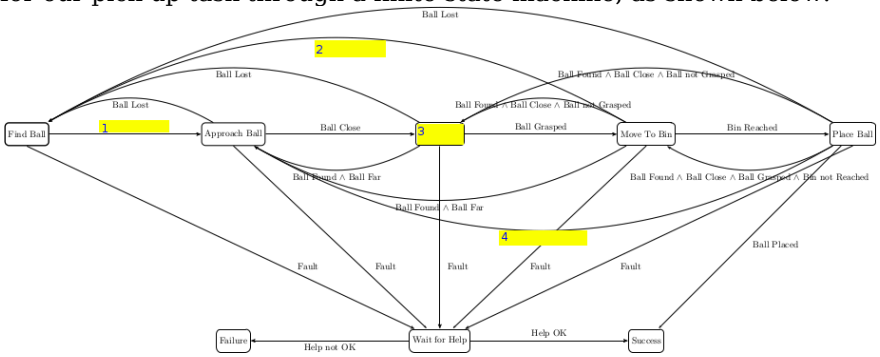


Figure 5.22 - Behavior Tree for Pick-Up Task

Here we've added the behavior that if the robot cannot complete the task, it can ask a kind human for help. Take a few minutes to walk through the behavior tree and convince yourself it encodes the desired task behavior.

a. We claimed behavior trees enable reactive behavior. Let's explore that. Imagine we have a robot executing our behavior tree. As the robot is executing the action "Approach Bin", a rude human knocks the ball out of the robot's hand. The ball rolls to a position that the robot can still see, but is quite far away. Following the logic of the behavior tree, what condition fails and what action is executed because of this?
b. Another way to mathematically model computation is through finite state machines (FSM). Finite state machines are composed of states, transitions and events. We can encode the behavior of our behavior tree for our pick-up task through a finite state machine, as shown below:



We have left a few states and transitions blank. Your task is to fill in those 4 values such that the finite state machine produces the same behavior as the behavior tree.

From this simple task we can see that while finite state machines can often be easy to understand and intuitive, they can quickly get quite messy!

c. Throughout this chapter and the lectures we've discussed a bin picking task. Our goal is to now capture the behavior of the bin-picking task using a behavior tree. We want to encode the following behavior: while Bin 1 is not empty, we are going to select a feasible grasp within the bin, execute our grasp and transport what's within our grasp to Bin 2. If there are still objects within Bin 1, but we cannot find a feasible grasp, assume we have some action that shakes the bin, hence randomizing the objects and (hopefully) making feasible grasps available.

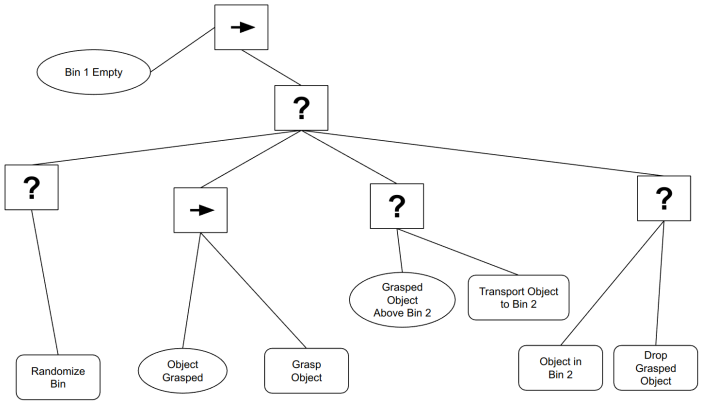
We define the following conditions:

- "Bin 1 empty?
- "Feasible Grasp Exists?"
- "Grasp selected?"
- "Grasped Object(s) above Bin 2?"
- "Object Grasped?"
- "Object(s) in Bin 2?"

And the following actions:

- "Drop grasped object(s) into Bin 2"
- "Grasp Object(s)"
- "Select Grasp"
- "Shake Bin"
- "Transport grasped object(s) to above Bin 2"

Using these conditions, actions and our two control flow nodes (Sequence and Fallback), we want you to draw a behavior tree that captures the behavior of our bin picking task. To get you started, we have included a "rough draft" behavior tree below, but it's broken in a number of places (e.g., some branches might be missing, some control flow nodes/actions/conditions might have the wrong type, etc.):



Your job is to identify all of the places where it's broken, and draw your own correct behavior tree with all the broken components fixed.

We claimed behavior trees are modular. Their modularity comes from the fact that we can write reusable snippets of code that define actions and conditions and then compose them using behavior trees. For example, we can imagine that the "Select Grasp" action from part (c) is implemented using the antipodal grasping mechanism we have built up throughout this chapter. And the "Transport grasped object(s) to above Bin 2" action could be implemented using the pick and place tools we developed in Chapter 3.

Source: The scenario and figures for parts (a) and (b) of this exercise are inspired by material in [15]

Exercise 5.6 (Simulation Tuning)

For this exercise, you will explore how to use and debug a physics simulator. In particular, you will inspect the contact forces of two interpenetrating boxes and examine discontinuities that arise due to point contact modeling. You will learn to debug and engineer collision geometries and physical parameter tuning to enable a sliding-block simulation to reach a desired final state. You will work both in as well as some written problems inside. There are 8 subproblems in this notebook, some involve coding and others involve answering written questions -- remember

to do them all!

References

1. Berk Calli and Arjun Singh and James Bruce and Aaron Walsman and Kurt Konolige and Siddhartha Srinivasa and Pieter Abbeel and Aaron M Dollar, "Yale-{CMU}-{B}erkeley dataset for robotic manipulation research", *The International Journal of Robotics Research*, vol. 36, no. 3, pp. 261--268, 2017.

2. Gregory Izatt and Russ Tedrake, "Generative Modeling of Environments with Scene Grammars and Variational Inference", *Proceedings of the 2020 IEEE International Conference on Robotics and Automation (ICRA)* , 2020. [link]

3. Ryan Elandt and Evan Drumwright and Michael Sherman and Andy Ruina, "A pressure field model for fast, robust approximation of net contact force and moment between nominally rigid objects", , pp. 8238-8245, 11, 2019.

4. Domenico Prattichizzo and Jeffrey C Trinkle, "Grasping", *Springer Handbook of Robotics* , pp. 671-700, 2008.

5. Hongkai Dai, "Robust multi-contact dynamical motion planning using contact wrench set", PhD thesis, Massachusetts Institute of Technology, 2016. [link]

6. Andreas ten Pas and Marcus Gualtieri and Kate Saenko and Robert Platt, "Grasp pose detection in point clouds", *The International Journal of Robotics Research*, vol. 36, no. 13-14, pp. 1455--1473, 2017.

7. Jeffrey Mahler and Jacky Liang and Sherdil Niyaz and Michael Laskey and Richard Doan and Xinyu Liu and Juan Aparicio Ojea and Ken Goldberg, "Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics", *arXiv preprint arXiv:1703.09312*, 2017.

8. Andy Zeng and Shuran Song and Kuan-Ting Yu and Elliott Donlon and Francois R Hogan and Maria Bauza and Daolin Ma and Orion Taylor and Melody Liu and Eudald Romo and others, "Robotic pick-and-place of novel objects in clutter with multi-affordance grasping and cross-domain image matching", *2018 IEEE international conference on robotics and automation (ICRA)* , pp. 1--8, 2018.

9. Craig M Shakarji, "Least-squares fitting algorithms of the NIST algorithm testing system", *Journal of research of the National Institute of Standards and Technology*, vol. 103, no. 6, pp. 633, 1998.

10. Radu Bogdan Rusu and Steve Cousins, "{3D is here: Point Cloud Library (PCL)}", *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* , May 9-13, 2011.

11. Radu Bogdan Rusu, "Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments", PhD thesis, Institut für Informatik der Technischen Universität München, 2010.

12. R. A. Brooks, "Elephants Don't Play Chess", *Robotics and Autonomous Systems*, vol. 6, pp. 3--15,, 1990.

13. Rodney A. Brooks, "Intelligence Without Reason", , no. 1293, April, 1991.

14. Rodney A. Brooks, "Intelligence without representation", *Artificial Intelligence*, vol. 47, pp. 139-159, 1991.

15. Michele Colledanchise and Petter Ogren, "Behavior trees in robotics and AI: An introduction", CRC Press , 2018.

Chapter 6
Mobile Manipulation

Coming soon!

Chapter 7
Motion Planning

There are a few more essential skills that we need in our toolbox. In this chapter, we will explore some of the powerful methods of kinematic trajectory motion planning.

I'm actually almost proud of making it this far into the notes without covering this topic yet. Writing a relatively simple script for the pose of the gripper, like we did in the bin picking chapter, really can solve a lot of interesting problems. But there are a number of reasons that we might want a more automated solution:

1. When the environment becomes more cluttered, it is harder to write such a simple solution, and we might have to worry about collisions between the arm and the environment as well as the gripper and the environment.
2. If we are doing "mobile manipulation" -- our robotic arms are attached to a mobile base -- then the robot might have to operate in many different environments. Even if the workspace is not geometrically complicated, it might still be different enough each time we reach that it requires automated (but possibly still simple) planning.
3. If the robot is operating in a simple known environment all day long, then it probably makes sense to optimize the trajectories that it is executing; we can often speed up the manipulation process significantly.

In fact, if you ran the clutter clearing demo, I would say that motion planning failures were the biggest limitation of that solution so far: the hand or objects could sometimes collide with the cameras or bins, or the differential-inverse kinematics strategy (which effectively ignored the joint angles) would sometime cause the robot to fold in on itself. In this chapter we'll develop the tools to make that much better!

I do need to make one important caveat. For motion planning in manipulation, lots of emphasis is placed on the problem of avoiding collisions. Despite having done some work in this field myself, I actually really dislike the problem formulation of collision-free motion planning. I think that on the whole, robots are too afraid of bumping into the world (because things still go wrong when they do). I don't think humans are solving these complex geometric problems every time we reach... even when we are reaching in dense clutter. I actually suspect that we are very bad at solving them. I would much rather see robots that perform well even with very coarse / approximate plans for moving through a cluttered environment, that are not afraid to make incidental contacts, and that can still accomplish the task when they do!

7.1 Inverse Kinematics

The goal of this chapter is to solve for motion trajectories. But I would argue that if you really understand how to solve inverse kinematics, then you've got most of what you need to plan trajectories.

We know that the forward kinematics give us a (nonlinear) mapping from joint angles to e.g. the pose of the gripper: $X^G = f_{kin}(q)$. So, naturally, one would think that the problem of inverse kinematics (IK) is about solving for the inverse map, $q = f_{kin}^{-1}(X^G)$. But, like we did with differential inverse kinematics, I'd like to think about inverse kinematics as the more general problem of finding joint angles subject to a rich library of costs and constraints; and the space of possible kinematic constraints is indeed rich.

For example, when we were evaluating grasp candidates for bin picking, we had only a soft preference on the orientation of the hand relative to some antipodal grasp. In that case, specifying 6 DOF pose of the gripper and finding one set of joint angles which satisfies it exactly would have been an overly constrained specification. I would say that it's rare that we have only end-effector pose constraints to reason

about, we almost always have costs or constraints in joint space (like joint limits) and others in Cartesian space (like non-penetration constraints).

Click here to watch the video.

Figure 7.1 - We made extensive use of rich inverse kinematics specifications in our work on humanoid robots. The video above is an example of the interactive inverse kinematics interface (here to help us figure out how to fit the our big humanoid robot into the little Polaris). Here is another video of the same tool being used for the Valkyrie humanoid, where we do specify and end-effector pose, but we also add a joint-centering objective and static stability constraints [1, 2].

7.1.1 From end-effector pose to joint angles

With its obvious importance in robotics, you probably won't be surprised to hear that there is an extensive literature on inverse kinematics. But you may be surprised at how extensive and complete the solutions can get. The forward kinematics, f_{kin}, is a nonlinear function in general, but it is a very structured one. In fact, with rare exceptions (like if your robot has a helical joint, aka screw joint), the equations governing the valid Cartesian positions of our robots are actually *polynomial*. "But wait! What about all of those sines and cosines in my kinematic equations?" you say. The trigonometric terms come when we want to relate joint angles with Cartesian coordinates. In $\mathbb{R}^3$, for two points, A and B, on the same rigid body, the (squared) distance between them, $\|p^A - p^B\|^2$, is a constant. And a joint is just a polynomial constraint between positions on adjoining bodies, e.g. that they occupy the same point in Cartesian space. See [3] for an excellent overview.

Understanding the solutions to polynomial equations is the subject of algebraic geometry. There is a deep literature on kinematics theory, on symbolic algorithms, and on numerical algorithms. For even very complex kinematic topologies, such as four-bar linkages and Stewart-Gough platforms, we can count the number of solutions, and/or understand the continuous manifold of solutions. For instance, [3] describes a substantial toolbox for numerical algebraic geometry (based on homotopy methods) with impressive results on difficult kinematics problems.

While the algebraic-geometry methods are mostly targeted for offline global analysis, they are not designed for fast real-time inverse kinematics solutions needed in a control loop. The most popular tool these days for real-time inverse kinematics for six- or seven-DOF manipulators is a tool called "IKFast", described in Section 4.1 of [4], that gained popularity because of its effective open-source implementation. Rather than focus on completeness, IKFast uses a number of approximations to provide fast and numerically robust solutions to the "easy" kinematics problems. It leverages the fact that a six-DOF pose constraint on a six-DOF manipulator has a "closed-form" solution with a finite number of joint space configurations that produce the same end-effector pose, and for seven-DOF manipulators it adds a layer of sampling in the last degree of freedom on top of the six-DOF solver.

These explicit solutions are important to understand because they provide deep insight into the equations, and because they can be fast enough to use inside a more sophisticated solution approach. But the solutions don't provide the rich specification I advocated for above; in particular, they break down once we have inequality constraints instead of equality constraints. For those richer specifications, we will turn to optimization.

7.1.2 IK as constrained optimization

Rather than formulate inverse kinematics as

$$q = f_{kin}^{-1}(X^G),$$

let's consider solving the same problem as an optimization:

$$\min_q \quad |q - q_0|^2, \tag{1}$$

$$\text{subject to} \quad X^G = f_{kin}(q), \tag{2}$$

where q_0 is some comfortable nominal position. While writing the inverse directly is a bit problematic, especially because we sometimes have multiple (even infinite) solutions or no solutions. This optimization formulation is slightly more precise -- if we have multiple joint angles which achieve the same end-effector position, then we prefer the one that is closest to the nominal joint positions. But the real value of switching to the optimization perspective of the problem is that it allows us to connect to a rich library of additional costs and constraints.

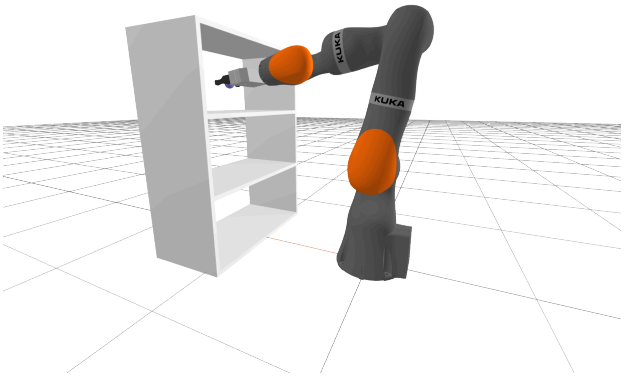


Figure 7.2 - A richer inverse kinematics problem: solve for the joint angles, q, that allow the robot to reach into the shelf and grab the object, while avoiding collisions.

We have already discussed the idea of solving *differential* inverse kinematics as an optimization problem. In that workflow, we started by using the pseudo-inverse of the kinematic Jacobian, but then graduated to thinking about the least-squares formulation of the inverse problem. The more general least-squares setting, we could add additional costs and constraints that would protect us from (nearly) singular Jacobians and could take into account additional constraints from joint limits, joint velocity limits, etc. We could even add collision avoidance constraints. Some of these constraints are quite nonlinear / nonconvex functions of the configuration q, but in the differential kinematics setting we were only seeking to find a small change Δq around the nominal configuration, so it was quite reasonable to make linear/convex approximations of these nonlinear/nonconvex constraints.

Now we will consider the full formulation, where we try to solve the nonlinear / nonconvex optimization directly, without any constraints on only making a small change to an initial q. This is a much harder problem computationally. Using powerful nonlinear optimization solvers like SNOPT, we are often able to solve the problems, even at interactive rates (the example below is quite fun). But there are no guarantees. It could be that a solution exists even if the solver returns "infeasible".

Of course, the differential IK problem and the full IK problem are closely related. In fact, you can think about the differential IK algorithm as doing one step of (projected) gradient descent or one-step of Sequential Quadratic Programming, for the full nonlinear problem.

Drake provides a nice InverseKinematics class that makes it easy to assemble many of the standard kinematic/multibody constraints into a `MathematicalProgram`. Take a minute to look at the constraints that are offered. You can add constraints on the relative position and/or orientation on two bodies, or that two bodies are more than some minimal distance apart (e.g. for non-penetration) or closer than some distance, and more. This is the way that I want you to think about the IK problem; it is an inverse problem, but one with a potentially very rich set of costs and constraints.

Example 7.1 (Interactive IK)

Despite the nonconvexity of the problem and nontrivial computational cost of evaluating the constraints, we can often solve it at interactive rates. I've assembled a few examples of this in the chapter notebook:

In the first version, I've added sliders to let you control the desired pose of the end-effector. This is the simple version of the IK problem, amenable to more explicit solutions, but we nevertheless solve it with our full nonlinear optimization IK engine (and it does include joint limit constraints). This demo won't look too different from the very first example in the notes, where you used teleop to command the robot to pick up the red brick. In fact, differential IK offers a fine solution to this problem, too.

In the second example, I've tried to highlight the differences between the nonlinear IK problem and the differential IK problem by adding an obstacle directly in front of the robot. Because both our differential IK and IK formulations are able to consume the collision-avoidance constraints, both solutions will try to prevent you from crashing the arm into the post. But if you move the target end-effector position from one side of the post to the other, the full IK solver can switch over to a new solution with the arm on the other side of the post, but the differential IK will never be able to make that leap (it will stay on the first side of the post, refusing to allow a collision).

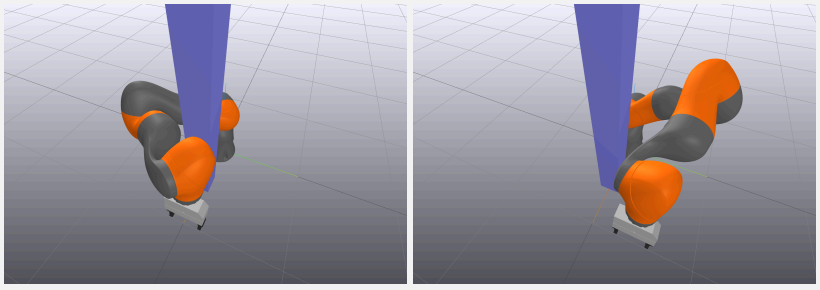


Figure 7.3 - As the desired end-effector position moves along positive y, the IK solver is able to find a new solution with the arm wrapped the other way around the post.

With great power comes great responsibility. The inverse kinematics toolbox allows you to formulate complex optimizations, but your success with solving them will depend partially on how thoughtful you are about choosing your costs and constraints. My basic advice is this:

1. Try to keep the objective (costs) simple; I typically only use the "joint-centering" quadratic cost on q. Putting terms that should be constraints into the cost as penalties leads to lots of cost-function tuning, which can be a nasty business.
2. Write minimal constraints. You want the set of feasible configurations to be as big as possible. For instance, if you don't need to fully constrain the orientation of the gripper, then don't do it.

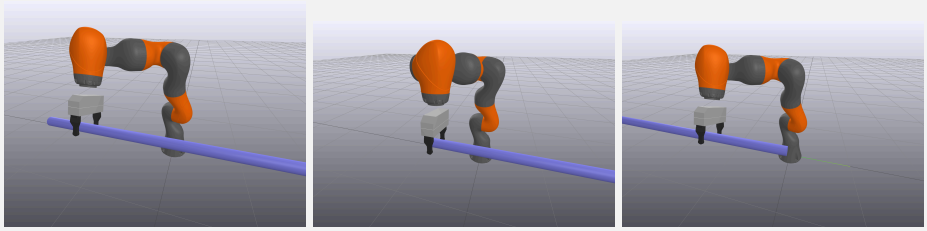
I'll follow-up with that second point using the following example.

Example 7.2 (Grasp the cylinder)

Let's use IK to grasp a cylinder (call it a hand rail). Suppose it doesn't matter where along the cylinder we grasp, nor the orientation at which we grasp it. Then we should write the IK problem using only the minimal version of those constraints.

In the notebook, I've coded up one version of this. I've put the cylinder's pose on

the sliders now, so you can move it around the workspace, and watch how the IK solver decides to position the robot. In particular, if you move the cylinder in $\pm y$, you'll see that the robot doesn't try to follow... until the hand gets to the end of the cylinder. Very nice!



One could imagine multiple ways to implement that constraint. Here's how I did it:

```
ik.AddPositionConstraint(
    frameB=gripper_frame, p_BQ=[0, 0.1, -0.02],
    frameA=cylinder_frame, p_AQ_lower=[0, 0, -0.5], p_AQ_upper=[0, 0, 0.5])
ik.AddPositionConstraint(
    frameB=gripper_frame, p_BQ=[0, 0.1, 0.02],
    frameA=cylinder_frame, p_AQ_lower=[0, 0, -0.5], p_AQ_upper=[0, 0, 0.5])
```

In words, I've defined two points in the gripper frame; in the notation of the `AddPositionConstraint` method they are ${}^{B}p^{Q}$. Recall the gripper frame is such that $[0, .1, 0]$ is right between the two gripper pads; you should take a moment to make sure you understand where $[0, .1, -0.02]$ and $[0, .1, 0.02]$ are. Our constraints require that both of those points should lie exactly on the center line segment of the cylinder. This was a compact way for me to leave the orientation around the cylinder axis as unconstrained, and capture the cylinder position constraints all quite nicely.

We've provided a rich language of constraints for specifying IK problems, including many which involve the kinematics of the robot and the geometry of the robot and the world (e.g., the minimum-distance constraints). Let's take a moment to appreciate the geometric puzzle that we are asking the optimizer to solve.

Example 7.3 (Visualizing the configuration space)

Let's return to the example of the iiwa reaching into the shelf. This IK problem has two major constraints: 1) we want the center of the target sphere to be in the center of the gripper, and 2) we want the arm to avoid collisions with the shelves. In order to plot these constraints, I've frozen three of the joints on the iiwa, leaving only the three corresponding motion in the $x - z$ plane.

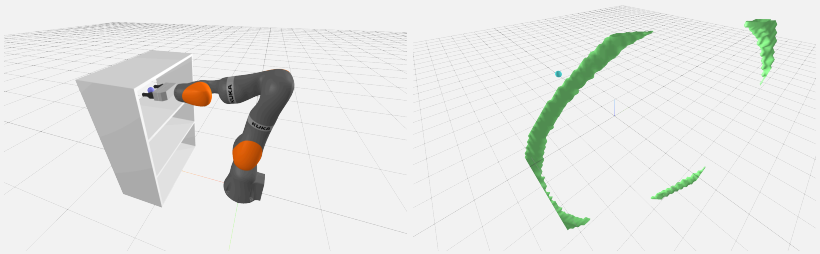


Figure 7.5 - The image on the right is a visualization of the "grasp the sphere" constraint in configuration space -- the x,y,z, axes in the visualizer correspond to the three joint angles of the planarized iiwa.

To visualize the constraints, I've sampled a dense grid in the three joint angles of the planarized iiwa, assigning each grid element to 1 if the constraint is satisfied or 0 otherwise, then run a marching cubes algorithm to extract an approximation of the true 3D geometry of this constraint in the configuration space. The "grasp the sphere" constraint produces the nice green geometry I've pictured above on the right; it is clipped by the joint limits. The collision-avoidance constraint, on the other hand, is quite a bit more complicated. To see that, you'd better open up this 3D visualization so you can navigate around it yourself. Scary!

Example 7.4 (Visualizing the IK optimization problem)

To help you appreciate the problem that we've formulated, I've made a visualization of the optimization landscape. Take a look at the landscape here first; this is only plotting a small region around the returned solution. You can use the Meshcat controls to show/hide each of the individual costs and constraints, to make sure you understand.

As recommended, I've kept the cost landscape (the *green* surface) to be simply the quadratic joint-centering cost. The constraints are plotted in *blue* when they are feasible, and *red* when they are infeasible:

- The joint limit constraint is just a simple "bounding-box" constraint here (only the red infeasible region is drawn for bounding box constraints, to avoid making the visualization too cluttered).
- The position constraint has three elements: for the x, y, and z positions of the end-effector. The y position constraint is trivially satisfied (all blue) because the manipulator only has the joints that move in the $x-z$ plane. The other two look all red, but if you turn off the y visualization, you can see two small strips of blue in each. That's the conditions in our tight position constraint.
- But it's the "minimum-distance" (non-collision) constraint that is the most impressive / scary of all. While we visualized the configuration space above, you can see here that visualizing the distance as a real-valued function reveals the optimization landscape that we give to the solver.

The intersection of all the blue regions here are what defined the configuration-space in the example above. All of the code for this visualization is available in the notebook, and you can find the exact formulation of the costs and constraints there:

You should also take a quick look at the full optimization landscape. This is the same set of curves as in the visualization above, but now it's plotted over the entire domain of joint angles within the joint limits. Nonlinear optimizers like SNOPT can be pretty amazing sometimes!

7.1.3 Global inverse kinematics

For unconstrained inverse kinematics with exactly six degrees of freedom, we have closed-form solutions. For the generalized inverse kinematics problem with rich costs and constraints, we've got a nonlinear optimization problem that works well in practice but is subject to local minima (and therefore can fail to find a feasible solution if it exists). If we give up on solving the optimization problem at interactive rates, is there any hope of solving the richer IK formulation robustly? Ideally to global optimality?

This is actually and extremely relevant question. There are many applications of inverse kinematics that work offline and don't have any strict timing requirements. Imagine if you wanted to generate training data to train a neural network to learn

your inverse kinematics; this would be a perfect application for global IK. Or if you want to do workspace analysis to see if the robot can reach all of the places it needs to reach in the workspace that you're designing for it, then you'd like to use global IK. Some of the motion planning strategies that we'll study below will also separate their computation into an offline "building" phase to make the online "query" phase much faster.

In my experience, general-purpose global nonlinear solvers -- for instance, based on mixed-integer nonlinear programming (MINLP) approaches or the interval arithmetic used in satisfiability-modulo-theories (SMT) solvers -- typically don't scale the complexity of a full manipulator. But if we only slightly restrict the class of costs and constraints that we support, then we can begin to make progress.

Drake provide an implementation of the GlobalInverseKinematics approach described in [5] using mixed-integer convex optimization. The solution times are on the order of a few seconds; it can solve a full constrained bimanual problem in well under a minute. [6] solves the narrow version of the problem (just end-effector poses and joint limits) using convex optimization via a hierarchy of semi-definite programming relaxations; it would be very interesting to understand how well this approach works the larger family of costs and constraints.

7.1.4 Inverse kinematics vs differential inverse kinematics

When should we use IK vs Differential IK? IK solves a more global problem, but is not guaranteed to succeed. It is also not guaranteed to make small changes to q as you make small changes in the cost/constraints; so you might end up sending large Δq commands to your robot. Use IK when you need to solve the more global problem, and the trajectory optimization algorithms we produce in the next section are the natural extension to producing actual q trajectories. Differential IK works extremely well for incremental motions -- for instance if you are able to design smooth commands in end-effector space and simply track them in joint space.

7.1.5 Grasp planning using inverse kinematics

In our first version of grasp selection using sampling, we put an objective that rewarded grasps that were oriented with the hand grasping from above the object. This was a (sometimes poor) surrogate for the problem that we really wanted to solve: we want the grasp to be achievable given a "comfortable" position of the robot. So a simple and natural extension of our grasp scoring metric would be to solve an inverse kinematics problem for the grasp candidate, and instead of putting costs on the end-effector orientation, we can use the joint-centering cost directly as our objective. Furthermore, if the IK problem returns infeasible, we should reject the sample.

There is a quite nice extension of this idea that becomes quite natural once we take the optimization view, and it is a nice transition to the trajectory planning we'll do in the next section. Imagine if the task requires us not only to pick up the object in clutter, but also to place the object carefully in clutter as well. In this case, a good grasp is finds a pose for the hand relative to the object that *simultaneously optimizes* both the pick configuration and the place configuration. This is actually quite natural to do with inverse kinematics; one can formulate an optimization problem with decision variables for both q_{pick} and q_{place}, with constraints enforcing that ${}^{O}X^{G_{pick}} = {}^{O}X^{G_{place}}$.

Once we add in all of our other rich costs and constraints, this becomes a quite sophisticated approach. In the dish-loading project at TRI, this approach proved to be very important. Both picking up a mug in the sink and placing it in the dishwasher rack are highly constrained, so we needed the simultaneous optimization to find successful grasps.

7.2 KINEMATIC TRAJECTORY OPTIMIZATION

Once you understand the optimization perspective of inverse kinematics, then you are well on your way to understanding kinematic trajectory optimization. Rather than solving multiple inverse kinematics problems independently, the basic idea now is to solve for a sequence of joint angles in the same optimization. Even better, let us define a parameterized joint trajectory, $q_\alpha(t)$, where α are parameters. Then a simple extension to our inverse kinematics problem would be to write something like

$$\min_{\alpha, T} \quad T, \tag{3}$$

$$\text{subject to} \quad X^{G_{start}} = f_{kin}(q_\alpha(0)), \tag{4}$$

$$X^{G_{goal}} = f_{kin}(q_\alpha(T)), \tag{5}$$

$$\forall t, \quad |\dot{q}_\alpha(t)| \leq v_{max}. \tag{6}$$

I read this as "find a trajectory, $q(t)$ for $t \in [0, T]$, that moves the gripper from the start to the goal in minimum time".

The last equation, (6), represents velocity limits; this is the only way we are telling the optimizer that the robot cannot teleport instantaneously from the start to the goal. Apart from this line which looks a little non-standard, it is almost exactly like solving two inverse kinematics problems jointly, except instead of having the solver take gradients with respect to q, we will take gradients with respect to α. This is easily accomplished using the chain rule.

7.2.1 Trajectory parameterizations

The interesting question, then, becomes how to do we actually parameterize the trajectory $q(t)$ with a parameter vector α? These says, you might think that $q_\alpha(t)$ could be a neural network that takes t as an input, offers q as an output, and uses α to represent the weights and biases of the network. Of course you could, but for inputs with a scalar input like this, we often take much simpler and sparser parameterizations, often based on polynomials.

There are many ways one can parameterize a trajectory with polynomials. For example in *dynamic* motion planning, direct collocation methods uses piecewise-cubic polynomials to represent the state trajectory, and the pseudo-spectral methods use Lagrange polynomials. In each case, the choice of basis functions is made so that algorithm can leverage a particular property of the basis. In dynamic motion planning, a great deal of focus is on the integration accuracy of the dynamic equations to ensure that we obtain feasible solutions to the dynamic constraints..

When we are planning the motions of our fully-actuated robot arms, we typically worry less about dynamic feasibility, and focus instead on the kinematics. For *kinematic* trajectory optimization, the so-called B-spline trajectory parameterization has a few particularly nice properties that we can leverage here:

- The derivative of a B-spline is still a B-spline (of one less degree), with coefficients that are linear in the original coefficients.
- The bases themselves are non-negative and sparse. This gives the coefficients of the B-spline polynomial, which are referred to as the *control points*, a strong geometric interpretation.
- In particular, the entire trajectory is guaranteed to lie inside the convex hull of the active control points (the control points who's bases are not zero).

Taken together this means that we can optimize over finitely parameterized trajectories, but use the convex hull property to ensure that limits on the joint positions and any of its derivatives are satisfied $\forall t \in [0, T]$ using *linear* constraints. This sort of guarantee is much more costly to obtain using most other polynomial bases.

Note that B-splines are closely related to Bézier curves. But the "B" in "B-spline"

actually just stands for "basis" (no, I'm not kidding) and "Bézier-splines" are slightly different.

7.2.2 Optimization algorithms

The default `KinematicTrajectoryOptimization` class in Drake optimizes a trajectory defined using a B-spline to represent a path, $r(s)$ over the interval $s \in [0, 1]$, plus an additional scalar decision variable corresponding to the trajectory duration, T. The final trajectory combines the path with the time-rescaling: $q(t) = r(t/T)$. This is a particularly nice way to represent a trajectory of unknown duration, and has the excellent feature that the convex hull property can still be used. Velocity constraints are still linear; constraints on acceleration and higher derivatives do become nonlinear, but if satisfied they still imply strict bounds $\forall t \in [0, T]$.

Since the `KinematicTrajectoryOptimization` is written using Drake's `MathematicalProgram`, by default it will automatically select what we think is the best solver given the available solvers. If the optimization has only convex costs and constraints, it will be dispatched to a convex optimization solver. But most often we add in nonconvex costs and constraints from kinematics. Therefore in most cases, the default solver would again be the SQP-solver, SNOPT. You are free to experiment with others!

One of the most interesting set of constraints that we can add to our kinematic trajectory optimization problem is the MinimumDistanceConstraint; when the minimum distance between all potential collision pairs is greater than zero then we have avoided collisions. Please note, though, that these collision constraints can only be enforced at discrete samples, $s_i \in [0, 1]$, along the path. *They do not guarantee that the trajectory is collision free* $\forall t \in [0, T]$. It required very special properties of the derivative constraints to leverage the convex hull property; we do not have them for more general nonlinear constraints. A common strategy is to add constraints at some modest number of samples along the interval during optimization, then to check for collisions more densely on the optimized trajectory before executing it on the robot.

Example 7.5 (Kinematic trajectory optimization for moving between shelves)

As a warm-up, I've provided a simple example of the planar iiwa reaching from the top shelf into the middle shelf.

If you look carefully at the code, I actually had to solve this trajectory optimization twice to get SNOPT to return a good solution (unfortunately, since it is a local optimization, this can happen). For this particular problem, the strategy that worked was to solve once without the collision avoidance constraint, and then use that trajectory as an initial guess for the problem with the collision avoidance constraint.

Another thing to notice in the code is the "visualization callback" that I use to plot a little blue line for the trajectory as the optimizer is solving. Visualization callbacks are implemented by e.g. telling the solver about a cost function that depends on all of the variables, and always returns zero cost; they get called every time the solver evaluates the cost functions. What I've done here can definitely slow down the optimization, but it's an excellent way to get some intuition about when the solver is "struggling" to numerically solve a problem. I think that the people / papers in this field with the fastest and most robust solvers are highly correlated with people who spend time visualizing and massaging the numerics of their solvers.

Example 7.6 (Kinematic trajectory optimization for clutter clearing)

We can use `KinematicTrajectoryOptimization` to do the planning for our clutter clearing example, too. This optimization was more robust, and did not require solving twice. I only seeded it with a trivial initial trajectory to avoid solutions where the robot tried to rotate 270 degrees around its base instead of taking the shorter path.

There are a number of related approaches to kinematic trajectory optimization in the motion planning literature, which differ either by their parameterization or by the solution technique (or both). Some of the more well-known include CHOMP [7], STOMP [8], and KOMO[9].

KOMO, for instance, is one of a wave of trajectory optimization techniques that use the Augmented Lagrangian method of transcribing a constrained optimization problem into an unconstrained problem, then using a simple but fast gradient-based solver[10]. Augmented-Lagrangian-based approaches appear to be the most popular and successful these days; I hope to provide a nice implementation in Drake soon!

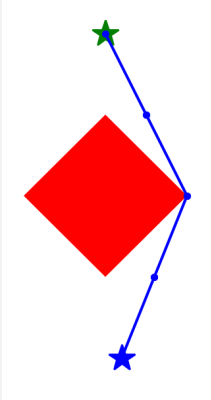
When kinematic trajectory optimizations succeed, they are an incredibly satisfying solution to the motion planning problem. They give a very natural and expressive language for specifying the desired motion (with needing to sample nonlinear constraints as perhaps the one exception), and they can be solved fast enough for online planning. The only problem is: they don't always succeed. Because they are based on nonconvex optimization, they are susceptible to local minima, and can fail to find a feasible path even when one exists.

Example 7.7 (Local minima in collision-free trajectory optimization)

Consider the extremely simple example of finding the shortest path from the start (blue star) to the goal (green star) in the image above, avoiding collisions with the red box. Avoiding even the complexity of B-splines, we can write an extremely simple optimization of the form:

$$\begin{aligned} \min_{q_0,\dots,q_N} \quad & \sum_{n=0}^{N-1} |q_{n+1} - q_n|_2^2 \\ \text{subject to} \quad & q_0 = q_{start} \\ & q_N = q_{goal} \\ & |q_n|_1 \geq 1 \qquad \forall n, \end{aligned}$$

where the last line is the collision-avoidance constraint saying that each sample point has to be *outside* of the ℓ_1-ball.



Once a nonlinear solver is considering paths that go right around the obstacle, it is very unlikely to find a solution that goes left around the obstacle, because the solution would have to get worse (violate the collision constraint) before it gets better.

To deal with this limitation, the field of collision-free motion planning has trended heavily towards sampling-based methods.

7.3 Sampling-based motion planning

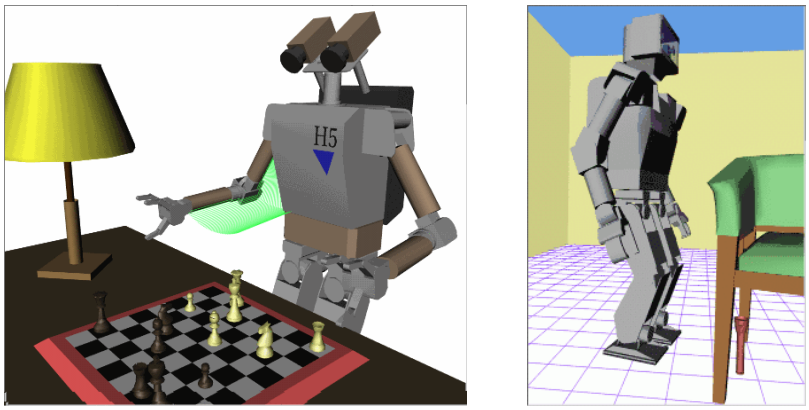


Figure 7.7 - Some incredible early (circa 2002) sampling-based motion planning results from James Kuffner (hover over the image to animate). These are kinematically complex and quite high dimensional.

The Open Motion Planning Library. We have our own implementations in Drake that are optimized for our collision engines.

7.3.1 Rapidly-exploring random trees (RRT)

Example 7.8 (The basic RRT)

Example 7.9 (The RRT "Bug Trap")

7.3.2 The Probabilistic Roadmap (PRM)

7.3.3 Post-processing

anytime b-spline smoother

7.4 Time-optimal path parameterizations

7.5 Graphs of Convex Sets

7.6 Exercises

Exercise 7.1 (Door Opening)

For this exercise, you will implement a optimization-based inverse kinematics solver to open a cupboard door. You will work exclusively in . You will be asked to complete the following steps:

a. Write down the constraints on the IK problem of grabbing a cupboard handle.
b. Formalize the IK problem as an instance of optimization.
c. Implement the optimization problem using MathematicalProgram.

Exercise 7.2 (RRT Motion Planning)

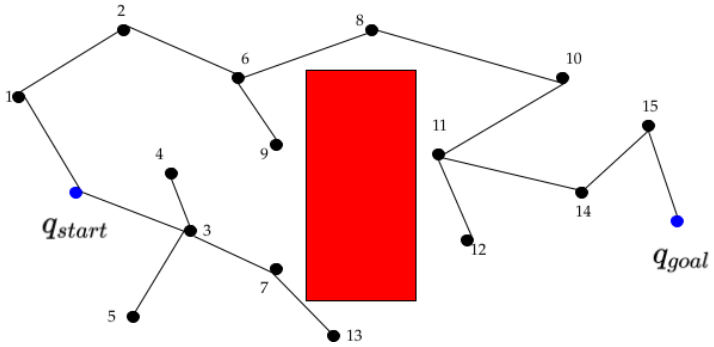
For this exercise, you will implement and analyze the RRT algorithm introduced in class. You will work exclusively in . You will be asked to complete the following steps:

a. Implement the RRT algorithm for the Kuka arm.
b. Answer questions regarding its properties.

Exercise 7.3 (Improving RRT Path Quality)

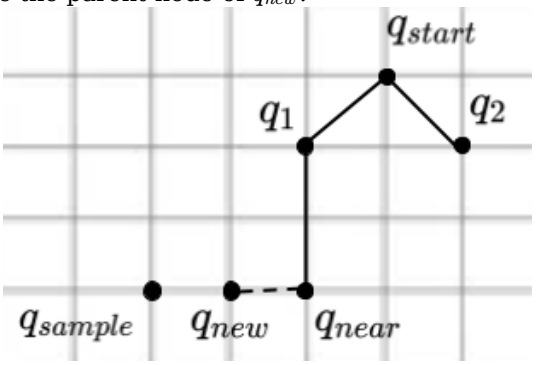
Due to the random process by which nodes are generated, the paths output by RRT can often look somewhat jerky (the "RRT dance" is the favorite dance move of many roboticists). There are many strategies to improve the quality of the paths and in this question we'll explore two. For the sake of this problem, we'll assume path quality refers to path length, i.e. that the goal is to find the shortest possible path, where distance is measured as Euclidean distance in joint space.

a. One strategy to improve path quality is to post-process paths via "shortcutting", which tries to replace existing portions of a path with shorter segments [11]. This is often implemented with the following algorithm: 1) Randomly select two non-consecutive nodes along the path. 2) Try to connect them with a RRT's extend operator. 3) If the resulting path is better, replace the existing path with the new, better path. Steps 1-3 are repeated until a termination condition (often a finite number of steps or time). For this problem, we can assume that the extend operator is a straight line in joint space. Consider the graph below, where RRT has found a rather jerky path from q_{start} to q_{goal}. There is an obstacle (shown in red) and q_{start} and q_{goal} are highlighted in blue (disclaimer: This graph was manually generated to serve as an illustrative example).



Name one pair of nodes for which the shortcutting algorithm would result in a shorter path (i.e. two nodes along our current solution path for which we could produce a shorter path if we were to directly connect them). You should assume the distance metric is the 2D Euclidean distance.

b. Shortcutting as a post-processing technique, reasons over the existing path and enables local "re-wiring" of the graph. It is a heuristic and does not, however, guarantee that the tree will encode the shortest path. To explore this, let's zoom in one one iteration of RRT (as illustrated below), where q_{sample} is the randomly generated configuration, q_{near} was the closest node on the existing tree and q_{new} is the RRT extension step from q_{near} in the direction of q_sample. When the standard RRT algorithm (which you implemented in a previous exercise) adds q_{new} to the tree, what node is its parent? If we wanted our tree to encode the shortest path from the starting node, q_{start}, to each node in the tree, what node should be the parent node of q_{new}?



This idea of dynamically "rewiring" to discover the minimum cost path (which for us is the shortest distance) is a critical aspect of the asymptotically optimal variant of RRT known as RRT* [12]. As the number of samples tends towards infinity RRT* finds the optimal path to the goal! This is unlike "plain" RRT, which is provably suboptimal (the intuition for this proof is that RRT "traps" itself because it cannot find better paths as it searches).

Exercise 7.4 (Decomposing Obstacle-Free Space with Convex Optimization)

For this exercise, you will implement part of the IRIS algorithm [13], which is used to compute large regions of obstacle-free space through a series of convex optimizations. These regions can be used by various planning methods that search for trajectories from start to goal while remaining collision-free. You will work exclusively in . You will be asked to complete the following steps:

a. Implement a QP that finds the closest point on an obstacle to an ellipse in free-space.
b. Implement the part of the algorithm that searches for a set of hyperplanes that separate a free-space ellipse from all the obstacles.

References

1. Maurice Fallon and Scott Kuindersma and Sisir Karumanchi and Matthew Antone and Toby Schneider and Hongkai Dai and Claudia P\'{e}rez D'Arpino and Robin Deits and Matt DiCicco and Dehann Fourie and Twan Koolen and Pat Marion and Michael Posa and Andr\'{e}s Valenzuela and Kuan-Ting Yu and Julie Shah and Karl Iagnemma and Russ Tedrake and Seth Teller, "An Architecture for Online Affordance-based Perception and Whole-body Planning", *Journal of Field Robotics*, vol. 32, no. 2, pp. 229-254, September, 2014. [link]

2. Pat Marion and Maurice Fallon and Robin Deits and Andr\'{e}s Valenzuela and Claudia P\'{e}rez D'Arpino and Greg Izatt and Lucas Manuelli and Matt Antone and Hongkai Dai and Twan Koolen and John Carter and Scott Kuindersma and Russ Tedrake, "Director: A User Interface Designed for Robot Operation With Shared Autonomy", *Journal of Field Robotics*, vol. 1556-4967, 2016. [link]

3. Charles W. Wampler and Andrew J. Sommese, "Numerical algebraic geometry and algebraic kinematics", *Acta Numerica*, vol. 20, pp. 469-567, 2011.

4. Rosen Diankov, "Automated Construction of Robotic Manipulation Programs", PhD thesis, Carnegie Mellon University, August, 2010.

5. Hongkai Dai and Gregory Izatt and Russ Tedrake, "Global inverse kinematics via mixed-integer convex optimization", *International Symposium on Robotics Research*, 2017. [link]

6. Pavel Trutman and Mohab Safey El Din and Didier Henrion and Tomas Pajdla, "Globally optimal solution to inverse kinematics of 7DOF serial manipulator", *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6012--6019, 2022.

7. Nathan Ratliff and Matthew Zucker and J. Andrew (Drew) Bagnell and Siddhartha Srinivasa, "{CHOMP}: Gradient Optimization Techniques for Efficient Motion Planning", *IEEE International Conference on Robotics and Automation (ICRA)* , May, 2009.

8. Mrinal Kalakrishnan and Sachin Chitta and Evangelos Theodorou and Peter Pastor and Stefan Schaal, "{STOMP}: Stochastic trajectory optimization for motion planning", *2011 IEEE international conference on robotics and automation* , pp. 4569--4574, 2011.

9. Marc Toussaint, "A tutorial on Newton methods for constrained trajectory optimization and relations to SLAM, Gaussian Process smoothing, optimal control, and probabilistic inference", *Geometric and numerical foundations of movements*, pp. 361--392, 2017.

10. Marc Toussaint, "A Novel Augmented Lagrangian Approach for Inequalities and Convergent Any-Time Non-Central Updates", , 2014.

11. R. Geraerts and M. Overmars, "A comparative study of probabilistic roadmap planners", *Algorithmic Foundations of Robotics V*, pp. 43--58,

2004.

12. S. Karaman and E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning", *Int. Journal of Robotics Research*, vol. 30, pp. 846--894, June, 2011.

13. Robin L H Deits and Russ Tedrake, "Computing Large Convex Regions of Obstacle-Free Space through Semidefinite Programming", *Proceedings of the Eleventh International Workshop on the Algorithmic Foundations of Robotics (WAFR 2014)* , 2014. [link]

CHAPTER 8
Manipulator Control

Over the last few chapters, we've developed a great initial toolkit for perceiving objects (or piles of objects) in a scene, planning grasps, and moving the arm to grasp. In the last chapter, we started designing motion planning strategies that could produce beautiful smooth joint trajectories for the robot that could move it quickly from one grasp to the next. As our desired trajectories get closer to the dynamic limits of the robot, then the control strategies that we use to execute them on the robot arm also need to get a bit more sophisticated.

Remember, too, that there is a lot more to manipulation than grasping! Even in the bin picking application, we already have some examples where our grasping-only strategy can be insufficient. Imagine that you look through the robot cameras into the box and see the following scene:

How in the world are we going to pick up that "Cheez-it" cracker box with our two-fingered gripper? (I know there are a few of you out there thinking about how easy that scene is for your suction gripper, but suction alone isn't going to solve all of our problems either.)

Click here to watch the video.

Figure 8.2 - Just for fun, I asked my daughter to try a similar task with the "two-fingered gripper" I scrounged from my kitchen. How are we to program something like that!

The term *nonprehensile manipulation* means "manipulation without grasping", and humans do it often. It is easy to appreciate this fact when we think about pushing an object that is too big to grasp (e.g. an office chair). But if you pay close attention, you will see that humans make use of strategies like sliding and environmental contact often, even when they are grasping. These strategies just come to the forefront in non-grasping manipulation.

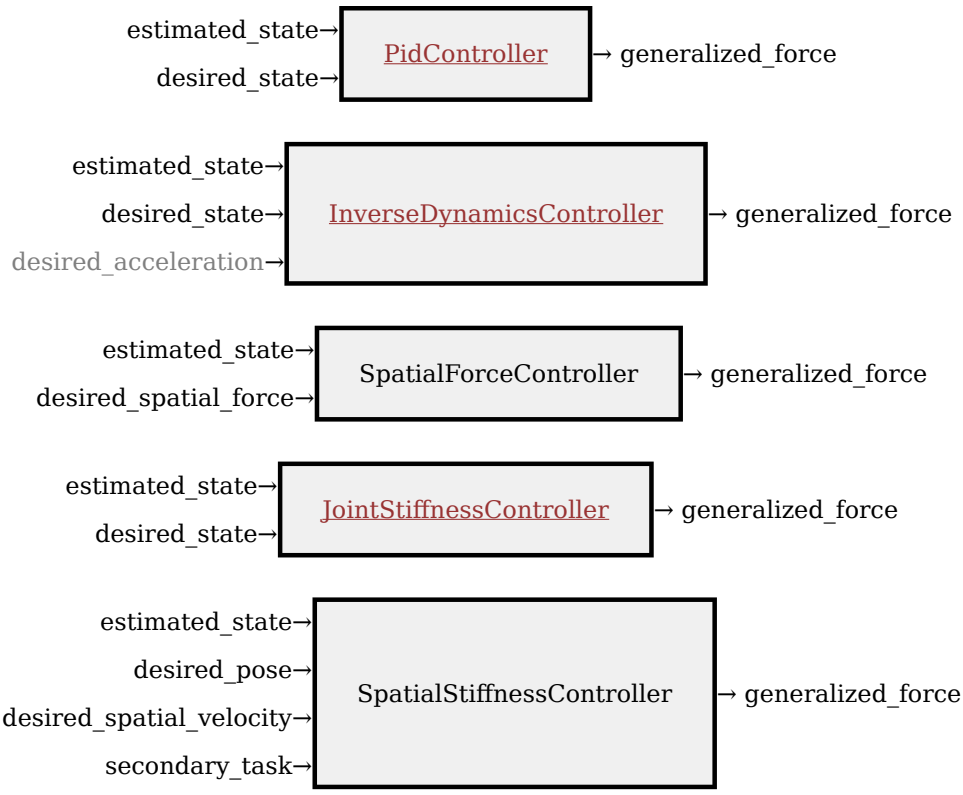
As we build up our toolkit for prehensile (grasping) and nonprehensile manipulation, one of the things that is missing from our discussion so far, which has been predominantly kinematic, has been thinking about forces. In addition to tracking motion planning trajectories, this chapter will also introduce control techniques that reason about contact forces. I hope that by the end you'll agree that we have pretty satisfying approaches to flipping up that box!

8.1 The Manipulator-Control Toolbox

When I talk about "manipulator control", I am discussing robot controllers that take (slightly) higher-level commands, like desired joint positions and velocities or spatial forces, and convert them into motor commands. Throughout this chapter we will assume that we can command *generalized forces*, which for revolute-joint robots are just joint torques. These controllers, by themselves, are not enough to complete any meaningful task -- they are reasoning about the robot itself but not about any of the objects in the environment. But they facilitate writing the rest of the control systems by providing the higher-level abstraction.

Typically the low-level controllers we will be discussing here are implemented in the firmware that runs directly on the robot arm (or its control cabinet). For the purposes of using that hardware, it's important to understand how they work and how you should set their parameters. In simulation we need to implement these controllers ourselves in order to model the robot.

Drake offers a number of manipulator control implementations. In this chapter, we will work through the most relevant ones for controlling a robotic arm:



By the end of the chapter, I hope you'll understand the differences and the essentials of how they all work.

8.2 Assume your robot is a point mass

As always in our battle against complexity, I want to find a setup that is as simple as possible (but no simpler)! Here is my proposal for the box-flipping example. First, I will restrict all motion to a 2D plane; this allows me to chop off two sides of the bin (for you to easily see inside), but also drops the number of degrees of freedom we have to consider. In particular, we can avoid the quaternion floating base coordinates, by adding a `PlanarJoint` to the box. Instead of using a complete

gripper, let's start even simpler and just use a "point finger". I've visualized it as a small sphere, and modeled two control inputs as providing force directly on the x and z coordinates of the point mass.

Figure 8.4 - The simple model: a point finger, a cracker box, and the bin all in 2D. The green arrows are the contact forces.

Even in this simple model, and throughout the discussions in this chapter, we will have two dynamic models that we care about. The first is the full dynamic model used for simulation: it contains the finger, the box, and the bin, and has a total of 5 degrees of freedom (x, z, θ for the box and x, z for the finger). The second is the robot model used for designing the controllers: this model has just the two degrees of freedom of the finger, and experiences unmodelled contact forces. By design, the equations of this second model are particularly simple:

$$\begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \dot{v} = m \begin{bmatrix} \ddot{x} \\ \ddot{z} \end{bmatrix} = \tau_g + \begin{bmatrix} u_x \\ u_z \end{bmatrix} + f^{F_c},$$

where m is the mass, τ_g is the gravitational "torque" which here is just $\tau_g = [0, -mg]^T$, u is the control input vector, and f^{F_c} is the Cartesian contact force applied to the finger, F, at c.

Just to make the notation clear, I've used f^{F_c} above to denote force *applied to the finger*, F, at point c. If we want to denote the same force *applied to the box*, B, at point c, then we'll denote it as f^{B_c}. Naturally, we have $f^{F_c} = -f^{B_c}$, because the forces must be equal and opposite. If we're not careful about this notation, the signs can get confusing below.

8.2.1 Trajectory tracking

Before we even make contact with the box, let's make sure we know how to move our robot finger around through the air. Assume that you've done some motion planning, perhaps using optimization, and have developed a beautiful desired trajectory, $q_d(t)$, that you want your finger to follow. Let's assume that $q_d(t)$ is twice differentiable. What generalized-force commands should you send to the robot to execute that trajectory?

One of the first and most common ways to execute the trajectory is with proportional-integral-derivative (PID) control, which we've discussed when we talked about position-controlled robots. The PidController uses the command

$$u(t) = K_p(q_d(t) - q(t)) + K_d(\dot{q}_d(t) - \dot{q}(t)) + K_i \int_0^t (q_d(t) - q(t))dt,$$

with K_p, K_d, and K_i being the position, velocity, and integral gains. Often these gains are diagonal matrices, making the command for each joint independent of the rest. Note that in manipulation, we tend to avoid the integral term, setting $K_i = 0$. You can imagine that if the robot is pushing up against the wall and is unable to achieve $q(t) == q_d(t)$, then the integral term will "wind-up", sending larger and larger commands to the robot until some fault is reached.

PD/PID control can be incredibly effective; it assumes almost nothing about the dynamics of the robot. But this assumption is also a limitation; if we do know something about the dynamics of the robot than we can achieve better tracking

performance. To make that clear, let's write the closed-loop dynamics under PD control...

Gravity compensation... For reasons that will become clear soon, this controller is available in Drake as the JointStiffnessController.

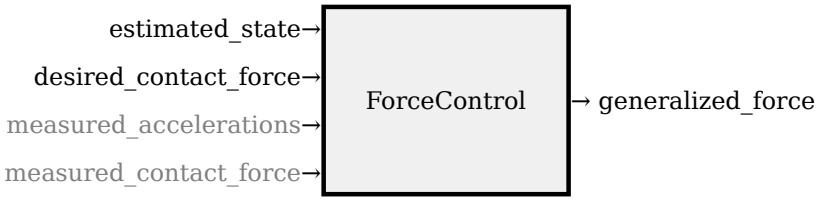
Inverse dynamics control...

Example 8.1 (Trajectory tracking for the point mass)

I've put together an extremely simple example to show the differences in tracking performance for the point mass using each of these control strategies.

8.2.2 (Direct) force control

In the case of direct force control, we implement a low-level controller that accepts the desired contact forces and produces generalized force inputs to the robot that try to make the actual contact forces match the desired.



What information do we need to regulate the contact forces? Certainly we need the desired contact force, $f^{F_c}_{desired}$. In general, we will need the robot's state (though in the immediate example, the dynamics of our point mass are not state dependent). But we also need to either (1) measure the robot accelerations (which we've repeatedly tried to avoid), (2) assume the robot accelerations are zero, or (3) provide a measurement of the contact force so that we can regulate it with feedback.

Let's consider the case where the robot is already in contact with the box. Let's also assume for a moment that the accelerations are (nearly) zero. This is actually not a horrible assumption for most manipulation tasks, where the movements are relatively slow. In this case, our equations of motion reduce to

$$f^{F_c} = -mg - u.$$

Our force controller implementation can be as simple as $u = -mg - f^{F_c}_{desired}$. Note that we only need to know the mass *of the robot* (not the box) to implement this controller.

What happens when the robot is not in contact? In this case, we cannot reasonably ignore the accelerations, and applying the same control results in $m\dot{v} = -f^{F_c}_{desired}$. That's not all bad. In fact, it's one of the defining features of force control that makes it very appealing. When you specify a desired force and don't get it, the result is accelerating the contact point in the (opposite) direction of the desired force. In practice, this (locally) tends to bring you into contact when you are not in contact.

Example 8.2 (Commanding a constant force)

Let's see what happens when we run a full simulation which includes not only the non-contact case and the contact case, but also the transition between the two (which involves collision dynamics). I'll start the point finger next to the box, and apply a constant force command requesting to get a horizontal contact force from

the box. I've drawn the x trajectory of the finger for different (constant) contact force commands.

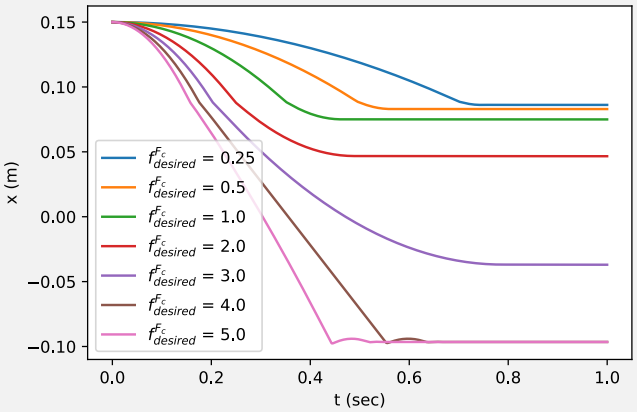


Figure 8.5 - This is a plot of the horizontal position of the finger as a function of time, given different constant desired force commands.

For all strictly positive desired force commands, the finger will accelerate at a constant rate until it collides with the box (at $x = 0.089$). For small $f^{F_c}_{desired}$, the box barely moves. For an intermediate range of $f^{F_c}_{desired}$, the collision with the box is enough to start it moving, but friction eventually brings the box and therefore also the finger to rest. For large values, the finger will keep pushing the box until it runs into the far wall.

Consider the consequences of this behavior. By commanding force, we can write a controller that will come into a nice contact with the box with essentially no information about the geometry of the box (we just need enough perception to start our finger in a location for which a straight-line approach will reach the box).

This is one of the reasons that researchers working on legged robots also like force control. On a force-capable walking robot, we might mimic position control during the "swing phase", to get our foot approximately where we are hoping to step. But then we switch to a force control mode to actually set the foot down on the ground. This can significantly reduce the requirements for accurate sensing of the terrain.

Example 8.3 (A force-based flip-up strategy)

Here is my proposal for a simple strategy for flipping up the box. Once in contact, we will use the contact force from the finger to apply a moment around the bottom left corner of the box to generate the rotation. But we'll add constraints to the forces that we apply so that the bottom corner does not slip.

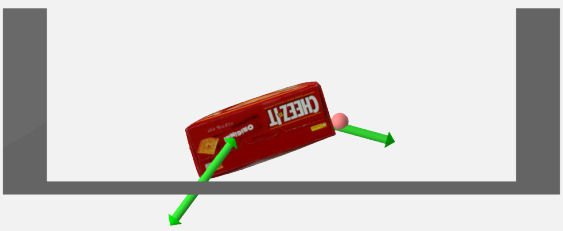


Figure 8.6 - Flipping the box. Click here for an animation of the controller we'll write in the example below. But it's worth running the code, where you can see the contact force visualization, too!

These conditions are very natural to express in terms of forces. And once again, we can implement this strategy with very little information about the box. The position of the finger will evolve naturally as we apply the contact forces. It's harder for me to imagine how to write an equally robust strategy using a (high-gain) position-controller finger; it would likely require many more assumptions about the geometry (and possibly the mass) of the box.

Let's encode the textual description above, describing the forces that are applied to the box. I'll use C for the contact frame between the finger and the box, with its normal pointing *into* the box normal to the surface, and A for the contact frame for the lower left corner of the box contacting the bin, with the normal pointing straight up in positive world z.

$$^B R^C = R_y(-\frac{\pi}{2}), \qquad ^W R^A = I,$$

where $R_y(\theta)$ is a rotation by θ around the y axis. Of course we also have the force of gravity, which is applied at the body center of mass (com):

$$f^{B_{com}}_{gravity,W} = mg.$$

As you can see, we'll make heavy use of the spatial force notation / spatial algebra described here.

The friction cone provides (linear inequality) constraints on the forces we want to apply.

$$\begin{aligned} f^{B_C}_{\text{finger},C_z} \geq 0, \qquad & |f^{B_C}_{\text{finger},C_x}| \leq \hat{\mu}_C f^{B_C}_{\text{finger},C_z}, \\ f^{B_A}_{\text{ground},A_z} \geq 0, \qquad & |f^{B_A}_{\text{ground},A_x}| \leq \hat{\mu}_A f^{B_A}_{\text{ground},A_z}. \end{aligned}$$

Please make sure you understand the notation. Within those constraints, we would like to rotate up the box.

To rotate the box about A, let's reason about the total torque being applied about A:

$$\tau^{B_A}_{total,W} = \tau^{B_A}_{gravity,W} + \tau^{B_A}_{ground,W} + \tau^{B_A}_{finger,W},$$

but we know that $\tau^{B_A}_{ground,W} = 0$ since the moment arm is zero. I have a goal here of not making too many assumptions about the mass and geometry of the box in our controller, so rather than try to regulate this torque perfectly, let's write

$$\tau^{B_A}_{total,W_y} = \tau^{B_A}_{finger,W_y} + \text{unknown terms}.$$

A reasonable control strategy in the face of these unmodeled terms is to use feedback control on the angle of the box (call it θ) which is the y component of $^W R^B$:

$$\tau^{B_A}_{finger_d,W_y} = \text{PID}(\theta_d, \theta), \qquad ^W R^B(\theta) = R_y(\theta),$$

where I've used θ_d as shorthand for the desired box angle and PID as shorthand for a simple proportional-integral-derivative term. Note that $\tau^{B_A}_{finger,W_y} \propto f^{B_C}_{finger,C_x}$, where the (constant) coefficient only depends on $^B\hat{p}^C$; it does not actually depend on $\hat{\theta}$.

To execute the desired PID control subject to the friction-cone constraints, we can use constrained least-squares:

$$\begin{aligned}\min_{f^{B_C}_{finger,C},\, f^{B_A}_{ground,A}} \quad & \tau^{B_A}_{finger,W_y} - \text{PID}(\theta_d, \hat{\theta})^2, \\ \text{subject to} \quad & f^{B_C}_{\text{finger},C_z} \geq 0, \qquad |f^{B_C}_{\text{finger},C_x}| \leq \hat{\mu}_C f^{B_C}_{\text{finger},C_z}, \\ & f^{B_A}_{\text{ground},A_z} \geq 0, \qquad |f^{B_A}_{\text{ground},A_x}| \leq \hat{\mu}_A f^{B_A}_{\text{ground},A_z}, \\ & f^{B_A}_{\text{ground},A} + \hat{f}^{B_A}_{gravity,A} + f^{B_A}_{finger,A} = 0.\end{aligned}$$

Note that the last line is still a linear constraint once $\hat{\theta}$ is given, despite requiring some spatial algebra operations. Implementing this strategy assumes:

- We have some approximation, $\hat{\theta}$, for the orientation of the box. We could obtain this from a point cloud normal estimation, or even from tracking the path of the fingers.
- We have conservative estimates of the coefficients of static friction between the wall and the box, $\hat{\mu}_A$, and between the finger and the box, $\hat{\mu}_C$, as well as the approximate location of the finger relative to the box corner, and the box mass, $\hat{m}$.

You should experiment and decide for yourself whether these estimates can be loose. I think that you'll find it's quite robust to knowing the mass and geometry of the box.

We have multiple controllers in this example. The first is the low-level force controller that takes a desired contact force and sends commands to the robot to attempt to regulate this force. The second is the higher-level controller that is looking at the orientation of the box and deciding which forces to request from the low-level controller.

Please also understand that this is not some unique or optimal strategy for box flipping. I'm simply trying to demonstrate that sometimes controllers which might be difficult to express otherwise can easily be expressed in terms of forces!

8.2.3 Indirect force control

There is a nice philosophical alternative to controlling the contact interactions by specifying the forces directly. Instead, we can program our robot to act like a (simple) mechanical system that reacts to contact forces in a desired way. This philosophy was described nicely in an important series of papers by Ken Salisbury introducing *stiffness control* [1] and then Neville Hogan introducing *impedance control* [2, 3, 4].

This approach is conceptually very nice. Imagine we were to walk up and push on the end-effector of the iiwa. With only knowledge of the parameters of the robot itself (not the environment), we can write a controller so that when we push on the end-effector, the end-effector pushes back (using the entire robot) as if you were pushing on, for instance, a simple spring-mass-damper system. Rather than attempting to achieve manipulation by moving the end-effector rigidly through a series of position commands, we can move the set points (and perhaps stiffness) of a soft virtual spring, and allow this virtual spring to generate our desired contact forces.

This approach can also have nice advantages for guaranteeing that your robot won't go unstable even in the face of unmodeled contact interactions. If the robot acts like a dissipative system and the environment is a dissipative system, then the entire system will be stable. Arguments of this form can ensure stability for even very complex system, building on the rich literature on *passivity theory* or more generally *Port-Hamiltonian* systems[5].

Our simple model with a point finger is ideal for understanding the essence of indirect force control. The original equations of motion of our system are

$$m \begin{bmatrix} \ddot{x} \\ \ddot{z} \end{bmatrix} = mg + u + f^{F_c}.$$

We can write a simple controller to make this system act, instead, like (passive) mass-spring-damper system:

$$m \begin{bmatrix} \ddot{x} \\ \ddot{z} \end{bmatrix} + b \begin{bmatrix} \dot{x} \\ \dot{z} \end{bmatrix} + k \begin{bmatrix} x - x_d \\ z - z_d \end{bmatrix} = f^{F_c},$$

with the rest position of the spring at (x_d, z_d). The controller that implements this follows easily; in the point finger case this has the familiar form of a proportional-derivative (PD) controller, but with an additional "feed-forward" term to cancel out gravity.

Technically, if we are just programming the stiffness and damping, as I've written here, then a controller of this form would commonly be referred to as "stiffness control", which is a subset of impedance control. We could also change the effective mass of the system; this would be impedance control in its full glory. My impression, though, is that the consensus in robot control experts is that changing the effective mass is most often considered not worth the complexity that comes from the extra sensing and bandwidth requirements.

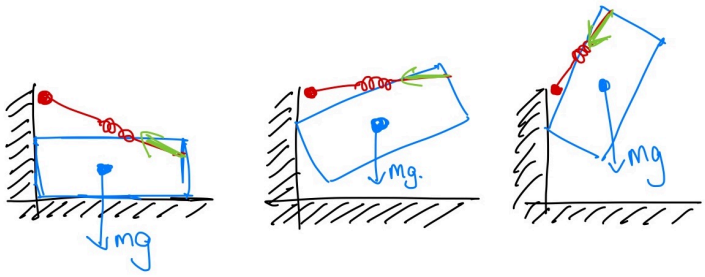
The literature on indirect force control has a lot of terminology and implementation details that are important to get right in practice. Your exact implementation will depend on, for instance, whether you have access to a force sensor and whether you can command forces/torque directly. The performance can vary significantly based on the bandwidth of your controller and the quality of your sensors. See e.g. [6] for a more thorough survey (and also some fun older videos), or [7] for a nice earlier perspective.

Example 8.4 (Teleop with stiffness control)

I didn't give you a teleop interface with direct force control; it would have been very difficult to use! Moving the robot by positioning the set points on virtual springs, however, is quite natural. Take a minute to try moving the box around, or even flipping it up.

To help your intuition, I've made the bin and the box slightly transparent, and added a visualization (in orange) of the virtual finger or gripper that you are moving with the sliders.

Let's embrace indirect force control to come up with another approach to flipping up the box. Flipping the box up in the middle of the bin required very explicit reasoning about forces in order to stay inside the friction cone in the bottom left corner of the box. But there is another strategy that doesn't require as precise control of the forces. Let's push the box into the corner, and *then* flip it up.



To make this one happen, I'd like to imagine creating a virtual spring -- you can think of it like a taut rubber band -- that we attach from the finger to a point near the wall just a little above the top left corner of the box. The box will act like a pendulum,

rotating around the top corner, with the rubber band creating the moment. At some point the top corner will slip, but the very same rubber band will have the finger pushing the box down from the top corner to complete the maneuver.

Consider the alternative of writing an estimator and controller that needed to detect the moment of slip and make a new plan. That is not a road to happiness. By only using our model of the robot to make the robot act like a different dynamical system at the point we can accomplish all of that!

Example 8.5 (A stiffness-control-based flip-up strategy)

This controller almost couldn't be simpler. I will just command a trajectory the move the virtual finger to just in front of the wall. This will push the box into contact and establish our bracing contact force. Then I'll move the virtual finger (the other end of our rubber band) up the wall a bit, and we can let mechanics take care of the rest!

So far we've made our finger act like two independent mass-spring-damper systems, one in x and the other in z. We even used the same stiffness, k, and damping, b, parameters for each. More generally, we can write stiffness and damping matrices, which we typically would call K_p and K_d, respectively:

$$m \begin{bmatrix} \ddot{x} \\ \ddot{z} \end{bmatrix} + K_d \begin{bmatrix} \dot{x} \\ \dot{z} \end{bmatrix} + K_p \begin{bmatrix} x - x_d \\ z - z_d \end{bmatrix} = f^{F_c}.$$

In order to emphasize the coordinate frame of these stiffness matrices, let's write exactly the same equation, realizing that with our point finger we have ${}^W p^F = [x, z]^T$, so:

$$m\, {}^W\dot{v}^F + K_d\, {}^W v^F + K_p \left({}^W p^F - {}^W p^{F_d} \right) = f^F.$$

Sometimes it can be convenient to express the stiffness (and/or damping) matrices in a different frame, A:

$$\begin{aligned} m\, {}^W\dot{v}^F + {}^W R^A K_d\, {}^W v_A^F + {}^W R^W K_p \left({}^W p_A^F - {}^W p_A^{F_d} \right) = \\ m\, {}^W\dot{v}^F + {}^W R^A K_d\, {}^A R^W\, {}^W v^F + {}^W R^A K_p\, {}^A R^W \left({}^W p^F - {}^W p^{F_d} \right) = f^F. \end{aligned}$$

8.2.4 Hybrid position/force control

There are a number of applications where we would like to explicitly command force in one direction, but command positions in another. One classic examples if you are trying to wipe or polish a surface -- you might care about the amount of force you are applying normal to the surface, but use position feedback to follow a trajectory in the directions tangent to the surface. In the simplest case, imagine controlling force in z and position in x for our point finger:

$$u = -mg + \begin{bmatrix} k_p(x_d - x) + k_d(\dot{x}_d - \dot{x}) \\ -f^F_{desired, W_z} \end{bmatrix}.$$

If want the forces/positions in a different frame, for instance the contact frame, C, then we can use

$$u = -mg + {}^W R^C \begin{bmatrix} k_p(p^{F_d}_{C_x} - p^F_{C_x}) + k_d(v^{F_d}_{C_x} - v^F_{C_x}) \\ -f^F_{desired, C_z} \end{bmatrix}. \quad (1)$$

By commanding the normal force, you not only have the benefit of controlling how hard the robot is pushing on the surface, but also gain some robustness to

errors in estimating the surface normal. If a position-controlled robot estimated the normal of a wall badly, then it might leave the wall entirely in one direction, and push extremely hard in the other. Having a commanded force in the normal direction would allow the position of the robot in that direction to become whatever is necessary to apply the force, and it will follow the wall.

Click here to watch the video.

Figure 8.8 - A simple example of hybrid position/force control; the robot needs to push down hard enough that the contact between the fingers and book do not slip, but gentle enough that the contacts between the book at the table *do* slide. This is naturally accomplished with force control in the world z. But moving the gripper/book in the x, y plane is more naturally described using position control. You can explore this example in more detail in an exercise.

The choice of position control or force control need not be a binary decision. We can simply apply both the position command (as in stiffness/impedance control) and a "feed-forward" force command:

$$u = -mg + K_p(p^{F_d} - p^F) + K_d(v^{F_d} - v^F) - f^F_{\text{feedforward}}.$$

Certainly this is only more general that the explicit position-or-force mode in Eq. (1); we could achieve the explicit position-or-force in this interface by the appropriate choices of K_p, K_d, and $f^F_{feedforward}$. As we'll see, this is quite similar to the interface provided by the iiwa (and many other torque-controlled robots).

8.3 The general case (using the manipulator equations)

Using the floating finger/gripper is a good way to understand the main concepts of force control without the details. But now it's time to actually implement those strategies using joint commands that we can send to the arm.

Our starting point is understanding that the equations of motion for a fully-actuated robotic manipulator have a very structured form:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} = \tau_g(q) + u + \sum_i J_i^T(q) f^{c_i}. \tag{2}$$

The left side of the equation is just a generalization of "mass times acceleration", with the mass matrix, M, and the Coriolis terms C. The right hand side is the sum of the (generalized) forces, with $\tau_g(q)$ capturing the forces due to gravity, u the joint torques supplied by the motors, and f^{c_i} is the Cartesian force due to the ith contact, where $J_i(q)$ is the ith "contact Jacobian". I introduced a version of these briefly when we described multibody dynamics for dropping random objects into the bin, and have more notes available here. For the purposes of the remainder of this chapter, we can assume that the robot is bolted to the table, so does not have any floating joints; I've therefore used $\dot{q}$ and $\ddot{q}$ instead of v and $\dot{v}$.

8.3.1 Trajectory tracking

Let us ignore the contact terms for a moment:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} = \tau_g(q) + u.$$

The *forward dynamics* problem, which we use during simulation, is to compute the accelerations, $\ddot{q}$, given the joint torques, u, (and $q, \dot{q}$). The *inverse dynamics* problem, which we can use in control, is to compute u given $\ddot{q}$ (and $q, \dot{q}$). As a system, it looks like

This system implements the controller:

$$u = M(q)\ddot{q}_d + C(q,\dot{q})\dot{q} - \tau_g(q),$$

so that the resulting dynamics are $M(q)\ddot{q} = M(q)\ddot{q}_d$. Since we know that $M(q)$ is invertible, this implies that $\ddot{q} = \ddot{q}_d$.

When we put the contact forces back in, In practice, we only have estimates of the dynamics (mass matrix, etc) and even the states q and $\dot{q}$. We will need to account for these errors when we determine the acceleration commands to send. For example, if we wanted to follow a desired joint trajectory, $q_0(t)$, we do not just differentiate the trajectory twice and command $\ddot{q}_0(t)$, but we could instead command e.g.

$$\ddot{q}_d = \ddot{q}_0(t) + K_p(q_0(t) - q) + K_d(\dot{q}_0(t) - \dot{q}).$$

One could also include an integral gain term.

The resulting `System` has one additional input port for the desired state:

This "inverse dynamics control" also appears in the literature with a few other names, such as "computed-torque control" and even the "feedforward-plus-feedback-linearizing control" [8, 11.4.1.3].

Check yourself: What is the difference between the joint stiffness control and the inverse dynamics control?

The closed-loop dynamics with the joint stiffness control (taking $\tau_{ff} = 0$) are given by

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + K_p(q - q_d) + K_d(\dot{q} - \dot{q}_d) = \tau_{ext},$$

where τ_{ext} summarized the results of (unmodeled) forces like contacts. The closed-loop dynamics with the inverse dynamics controllers are given by

$$\ddot{q} + K_p(q - q_d) + K_d(\dot{q} - \dot{q}_d) = M^{-1}(q)\tau_{ext}.$$

So the inverse dynamics controller writes the feedback law in the units of accelerations rather than forces; this would be similar to shaping the internal matrix into the diagonal matrix.

The ability to realize this cancellation in hardware, however, will be limited by the accuracy with which we estimate the model parameters, $\hat{M}(q)$ and $\hat{C}(q,\dot{q})$. The stiffness controller, on the other hand, achieves much of the desired performance but only attempts to cancel the gravity (and perhaps friction) terms; it does not shape the inertia.

8.3.2 Joint stiffness control

In practice, the way that we most often interface with the iiwa is through the its "joint-impedance control" mode, which is written up nicely in [9, 10]. For our purposes, we can view this as a stiffness control in joint space:

$$u = -\tau_g(q) + K_p(q_d - q) + K_d(\dot{q}_d - \dot{q}) + \tau_{ff},$$

where K_p, K_d are positive diagonal matrices, and τ_{ff} is a "feed-forward" torque. In practice the controller also includes some joint friction compensation[11], but I've left those friction terms out here for the sake of brevity. The controller does also do some shaping of the inertias (earning it the label "impedance control" instead of only "stiffness control"), but only the rotor inertias, and the user does not set these effective inertias. From the user perspective it should be viewed as stiffness control.

Check yourself: What is the difference between traditional position control with a PD controller and joint-stiffness control?

The difference in the algebra is quite small. A PD control would typically have the form

$$u = K_p(q_d - q) + K_d(\dot{q}_d - \dot{q}),$$

whereas stiffness control is

$$u = -\tau_g(q) + K_p(q_d - q) + K_d(\dot{q}_d - \dot{q}).$$

In other words, stiffness control tries to cancel out the gravity and any other estimated terms, like friction, in the model. As written, this obviously requires an estimated model (which we have for iiwa, but don't believe we have for most arms with large gear-ratio transmissions) and torque control. But this small difference in the algebra can make a profound difference in practice. The controller is no longer depending on error to achieve the joint position in steady state. As such we can turn the gains way down, and in practice have a much more compliant response while still achieving good tracking when there are no external torques.

8.3.3 Cartesian stiffness control

A better analogy for the control we were doing with the point finger example is to write a controller so that the robot acts like a simple dynamical system in the world frame. To do that, we have to identify a frame, E for "end-effector", on the robot where we want to impose these simple dynamics -- ideally the origin of this frame will be the expected point of contact with the environment. Following our treatment of kinematics and differential kinematics, we'll define the forward kinematics of this frame as:

$$p^E = f_{kin}(q), \qquad v^E = \dot{p}^E = J(q)\dot{q}, \qquad a^E = \ddot{p}^E = J(q)\ddot{q} + \dot{J}(q)\dot{q}. \tag{3}$$

We haven't actually written the second derivative before, but it follows naturally from the chain rule. Also, I've restricted this to the Cartesian positions for simplicity; one can think about the orientation of the end-effector, but this requires some care in defining the 3D stiffness in orientation (e.g. [12]).

One of the big ideas from manipulator control is that we can actually write the dynamics of the robot in the frame E, by parameterizing the joint torques as a translational force applied to body B at E, $f_u^{B_E}$: $u = J^T(q) f_u^{B_E}$. This comes from the principle of virtual work. By substituting this and the manipulator equations (2) into (3) and assuming that the only external contacts happen at E, we can write the dynamics:

$$M_E(q)\ddot{p}^E + C_E(q,\dot{q})\dot{q} = f_g^{B_E}(q) + f_u^{B_E} + f_{ext}^{B_E}, \tag{4}$$

where

$$M_E(q) = (JM^{-1}J^T)^{-1}, \qquad C_E(q,\dot{q}) = M_E\left(JM^{-1}C + \dot{J}\right), \qquad f_g^{B_E}(q) = M_E J M^{-1}\tau_g.$$

Now if we simply apply a controller analogous to what we used in joint space, e.g.:

$$f_u^{B_E} = -f_g^{B_E}(q) + K_p(p^{E_d} - p^E) + K_d(\dot{p}^{E_d} - \dot{p}^E),$$

then we can achieve the desired closed-loop dynamics *at the end-effector*:

$$M_E(q)\ddot{p}^E + C_E(q,\dot{q})\dot{q} + K_p(p^E - p^{E_d}) + K_d(\dot{p}^E - \dot{p}^{E_d}) = f_{ext}^{B_E}.$$

So if I push on the robot at the end-effector, the entire robot will move in a way that will make it feel like I'm pushing on a spring-damper system. Beautiful!

When $J(q)$ can be rank-deficient (for instance, if you have more degrees of freedom than you are trying to control at the end-effector), you'll also want to add some terms to similar to stabilize the null space of your Jacobian. Since we are operating here with torques instead of velocity commands, the natural analogue is to write the a joint-centering PD controller that operates in the nullspace of the end-effector control:

$$u = J^T f_u^{B_E} + [I - J^+ J][K_{p,joint}(q_0 - q) - K_{d,joint}\dot{q}].$$

This approach of programming the task-space dynamics with secondary joint-space objectives operating in the nullspace was developed in [13] and is known as *operational space control*. It can be generalized to a rich composition of *prioritized* task-space and joint-space control objectives; the prioritization is accomplished by putting lower-priority tasks in the nullspace of the primary task. Like the joint-centering differential inverse kinematics, these can also be understood through the lens of least-squares optimization.

8.3.4 Some implementation details on the iiwa

The implementation of the low-level impedance controllers has many details that are explained nicely in [9, 10]. In particular, the authors go to quite some length to implement the impedance law in the actuator coordinates rather than the joint coordinates (remember that they have an elastic transmission in between the two). I suspect there are many subtle reasons to prefer this, but they go to lengths to demonstrate that they can make a true passivity argument with this controller.

The iiwa interface offers a Cartesian impedance control mode. If we want high performance stiffness control in end-effector coordinates, then we should definitely use it! The iiwa controller runs at a much higher bandwidth (faster update rate) than the interface we have over the provided network API, and many implementation details that they have gone to great lengths to get right. But in practice we do not use it, because we cannot convert between joint impedance control and Cartesian impedance control without shutting down the robot. Sigh. In fact we cannot even change the stiffness gains nor the frame C (aka the "end-effector location") without stopping the robot. So we stay in joint impedance mode and command some Cartesian forces through τ_{ff} if we desire them. (If you are interested in the driver details, then I would recommend the documentation for the Franka Control Interface which is much easier to find and read, and is very similar to functionality provided by the iiwa driver.)

You might also notice that the interface we provide to the `ManipulationStation` takes a desired joint position for the robot, but not a desired joint velocity. That is because we cannot actually send a desired joint velocity to the iiwa controller. In practice, we believe that they are numerically differentiating our position commands to obtain the desired velocity, which adds a delay of a few control timesteps (and sometimes non-monotonic behavior). I believe that the reason why we aren't allowed to send our own joint velocity commands is to to make the passivity argument in their paper go through, and perhaps to make a simpler/safer API -- they don't want users to be able to send a series of inconsistent position and velocity commands (where the velocities are not actually the time derivatives of the positions).

Since the controller attempts to cancel gravitational terms, one can also tell the iiwa firmware about the lumped inertia of the gripper. This is set just once (in the pendant), and cannot be changed while the robot is moving; if you pick something

up (or even move the fingers), those changes in the gravitational terms will not be compensated in the controller.

Although the JointStiffnessController in Drake is the best model for the iiwa control stack, we commonly use the InverseDynamicsController in our simulations instead. This is for a fairly subtle reason -- the numerics are better. The effective stiffness of the differential equations to achieve comparable stiffness in the physics is smaller, and one can take bigger integration timesteps. On the iiwa hardware, we commonly use [800., 600, 600, 600, 400, 200, 200] Nm/rad as the stiffness values for the JointStiffnessController; but simulating with those requires small integration steps.

8.4 Putting it all together

8.5 Peg in hole

One of the classic problems for manipulator force control, inspired by robotic assembly, is the problem of mating two parts together under tight kinematic tolerances. Perhaps the cleanest and most famous version of this is the problem of inserting a (round) peg into a (round) hole. If your peg is square and your hole is round, I can't help. Interestingly, much of the literature on this topic in the late 1970's and early 1980's came out of MIT and the Draper Laboratory.

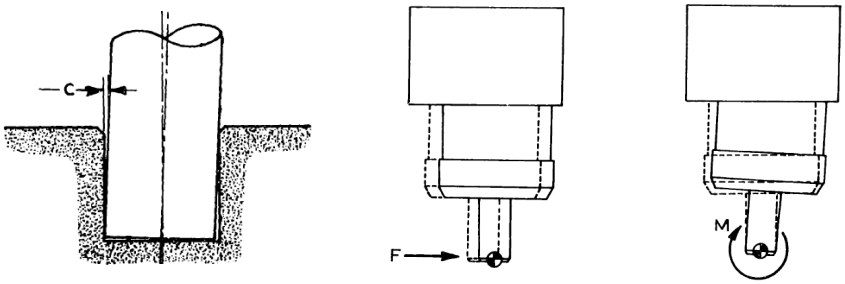


Figure 8.9 - (Left) the "peg in hole" insertion problem with kinematic tolerance c. (Middle) the desired response to a contact force. (Right) the desired response to a contact moment. Figures reproduced (with some symbols removed) from [14].

For many peg insertion tasks, it's acceptable to add a small chamfer to the part and/or the top of the hole. This allows small errors in horizontal alignment to be accounted for with a simple compliance in the horizontal direction. Misalignment in orientation is even more interesting. Small misalignments can cause the object to become "jammed", at which point the frictional forces become large and ambiguous (under the Coulomb model); we really want to avoid this. Again, we can avoid jamming by allowing a rotational compliance at the contact. I think you can see why this is a great example for force control!

The key insight here is that the rotational stiffness that we want should result in rotations around the part/contact, not around the gripper. We can program this with stiffness control; but for very tight tolerances the sensing and bandwidth requirements become a real limitation. One of the coolest results in force control is the realization that a clever physical mechanism can be used to produce a "remote-centered compliance" (RCC) completely mechanically, with infinite bandwidth and no sensing required [14]!

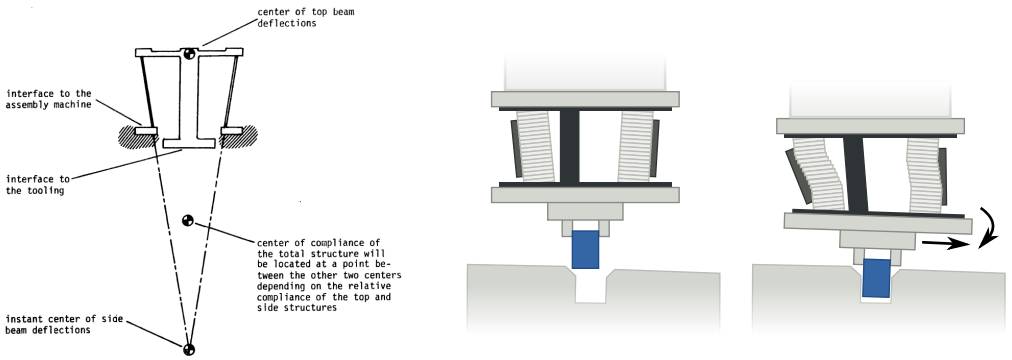


Figure 8.10 - A passive remote-center compliance device. (Left) concept schematic from [14]. (Right) schematic of a mechanical implementation; image credit for Arne Nordmann, 2008. See here for some example products, and here for a nice (older) video.

Interestingly, the peg-in-hole problem inspired incredibly important and influential ideas in mechanics and control, but also in motion planning [15]. The general ideas are quite relevant and applicable for a wide variety of tasks for which compliant contact is a reasonable strategy, such as opening doors, tool use, etc.

8.6 Exercises

Exercise 8.1 (Force and Position Control)

Suppose you are given a point-mass system with the following dynamics:

$$m\ddot{x} = u + f^{F_c}$$

where x is the position, u the input force applied on the point-mass, and f^{F_c} the contact force from the environment on point F applied at c. In order to control the position and force from the robot at the same time, the following controller is proposed:

$$u = \underbrace{k_p(x_d - x) - k_d\dot{x}}_{\text{feedback force for position control}} - \underbrace{f^{F_c}_{desired}}_{\text{feedforward force}}$$

Define the error for this system to be:

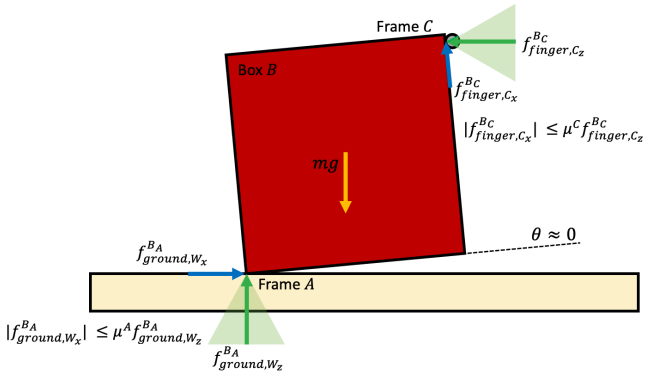
$$e = (x - x_d)^2 + (f^{F_c} - f^{F_c}_{desired})^2$$

a. Let's consider the case where our system is in free space (i.e. $f^{F_c} = 0$) and we want to exert a non-zero desired force ($f^{F_c}_{desired} \neq 0$) and drive our position to zero ($x_d = 0$). You wil show that this is not possible, i.e. we cannot achieve zero error for our desired force and desired position. You can show this by considering the steady-state error (i.e. set $\dot{x} = 0, \ddot{x} = 0$) as a function of the system dynamics, controller, desired position and desired force.
b. Now consider the system is in rigid body contact with a wall located at $x_d = 0$ and a desired non-zero force ($f^{F_c}_{desired} > 0$) is being commanded. Considering the steady-state error in this case, show that system can achieve zero error.

Exercise 8.2 (Box Flip-up)

Consider the initial phase of the box flip-up task from the example above, when

$\theta \approx 0$. Let B be the box, A be the pivot point of the box, and C be the contact point between the finger and box B. We will assume that the forces can still be approximated as axis-aligned, but the ground contact is only being applied at the pivoting point A.



a. If we do not push hard enough (i.e. $f^{B_C}_{finger,C_z}$ is too small), then we will not be able to create enough friction ($f^{B_C}_{finger,C_x}$) to counteract gravity. By summing the torques around the pivot point A, derive a lower bound for the required normal force $f^{B_C}_{finger,C_z}$, as a function of m, g, and μ^C. You may assume that the moment arm of gravity is half that of the moment arm on C, and that the box is square.
b. If we push too hard (i.e. $f^{B_C}_{finger,C_z}$ is too big), then the pivot point will end up sliding to the left. By writing the force balance along the horizontal axis, derive an upper bound for the required normal force $f^{B_C}_{finger,C_z}$, as a function of m, g, and μ^A. (HINT: It will be necessary to use the torque balance equation from before)
c. By combining the lower bounds and upper bounds from the previous problems, derive a relation for μ^A, μ^C that must hold in order for this motion to be possible. If $\mu^A \approx 0.25$ (wood-metal), how high does μ^C have to be in order for the motion to be feasible? What if $\mu^A = 1$ (concrete-rubber)?

Exercise 8.3 (Hybrid Force-Position Control)

For this exercise, you will analyze and implement a hybrid force-position controller to drag a book, as we saw from this video during lecture. You will work exclusively in . You will be asked to complete the following steps:

a. Analyze the conditions for this motion to be feasible.
b. Implement a control law to implement the motion

References

1. K. Salisbury, "Active stiffness control of a manipulator in {C}artesian coordinates", *Proc. of the 19th IEEE Conference on Decision and Control* , 1980.

2. Neville Hogan, "Impedance Control: An Approach to Manipulation. Part I -

Theory", *Journal of Dynamic Systems, Measurement and Control*, vol. 107, pp. 1--7, Mar, 1985.

3. Neville Hogan, "Impedance Control: An Approach to Manipulation. Part II - Implementation", *Journal of Dynamic Systems, Measurement and Control*, vol. 107, pp. 8--16, Mar, 1985.

4. Neville Hogan, "Impedance Control: An Approach to Manipulation. Part III - Applications", *Journal of Dynamic Systems, Measurement and Control*, vol. 107, pp. 17--24, Mar, 1985.

5. "Modeling and Control of Complex Physical Systems: The Port-Hamiltonian Approach", Springer-Verlag , 2009.

6. Luigi Villani and J De Schutterm, "9", in: Force Control, Springer *Handbook of Robotics* , 2008.

7. Daniel E Whitney, "Historical perspective and state of the art in robot force control", *The International Journal of Robotics Research*, vol. 6, no. 1, pp. 3--14, 1987.

8. Kevin M Lynch and Frank C Park, "Modern Robotics", Cambridge University Press , 2017.

9. Christian Ott and Alin Albu-Schaffer and Andreas Kugi and Gerd Hirzinger, "On the passivity-based impedance control of flexible joint robots", *IEEE Transactions on Robotics*, vol. 24, no. 2, pp. 416--429, 2008.

10. Alin Albu-Schaffer and Christian Ott and Gerd Hirzinger, "A unified passivity-based control framework for position, torque and impedance control of flexible joint robots", *The international journal of robotics research*, vol. 26, no. 1, pp. 23--39, 2007.

11. Alin Albu-Schaffer and Gerd Hirzinger, "A globally stable state feedback controller for flexible joint robots", *Advanced Robotics*, vol. 15, no. 8, pp. 799--814, 2001.

12. H.J. Terry Suh and Naveen Kuppuswamy and Tao Pang and Paul Mitiguy and Alex Alspach and Russ Tedrake, "{SEED}: Series Elastic End Effectors in 6D for Visuotactile Tool Use", *Under review* , 2022. [link]

13. Oussama Khatib, "A Unified Approach for Motion and Force Control of Robot Manipulators: The Operational Space Formulation", *IEEE Journal of Robotics and Automation*, vol. 3, no. 1, pp. 43-53, February, 1987.

14. Samuel Hunt Drake, "Using compliance in lieu of sensory feedback for automatic assembly.", PhD thesis, Massachusetts Institute of Technology, 1978.

15. Tomas Lozano-Perez and Matthew Mason and Russell Taylor, "Automatic synthesis of fine-motion strategies for robots", *The International Journal of Robotics Research*, vol. 3, no. 1, pp. 3--24, 1984.

Chapter 9
Object Detection and Segmentation

Our study of geometric perception gave us good tools for estimating the pose of a known object. These algorithms can produce highly accurate estimates, but are still subject to local minima. When the scenes get more cluttered/complicated, or if we are dealing with many different object types, they really don't offer an adequate solution by themselves.

Deep learning has given us data-driven solutions that complement our geometric approaches beautifully. Finding correlations in massive datasets has proven to be a fantastic way to provide practical solutions to these more "global" problems like detecting whether the mustard bottle is even in the scene, segmenting out the portion of the image / point cloud that is relevant to the object, and even in providing a rough estimate of the pose that could be refined with a geometric method.

There are many sources of information about deep learning on the internet, and I have no ambition of replicating nor replacing them here. But this chapter does being our exploration of deep perception in manipulation, and I feel that I need to give just a little context.

9.1 Getting to big data

9.1.1 Crowd-sourced annotation datasets

The modern revolution in computer vision was unquestionably fueled by the availability of massive annotated datasets. The most famous of all is ImageNet, which eclipsed previous datasets with the number of images and the accuracy and usefulness of the labels[1]. Fei-fei Li, who led the creation of ImageNet has been giving talks that give some nice historical perspective on how ImageNet came to be. Here is one (slightly) tailored to robotics and even manipulation; you might start here.

[1] describes the annotations available in ImageNet:

> ... annotations fall into one of two categories: (1) image-level annotation of a binary label for the presence or absence of an object class in the image, e.g., "there are cars in this image" but "there are no tigers," and (2) object-level annotation of a tight bounding box and class label around an object instance in the image, e.g., "there is a screwdriver centered at position (20,25) with width of 50 pixels and height of 30 pixels".

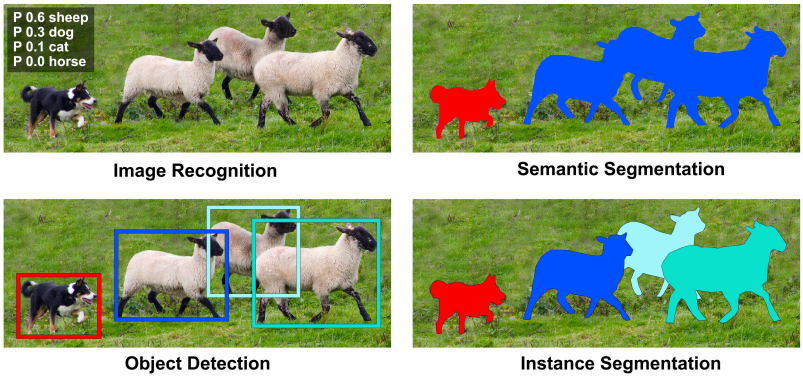


Figure 9.1 - A sample annotated image from the COCO dataset, illustrating the difference between image-level annotations, object-level annotations, and segmentations at the class/semantic- or instance- level..

The COCO dataset similarly enabled pixel-wise *instance-level* segmentation [2], where distinct instances of a class are given a unique label (and also associated with the class label). COCO has fewer object categories than ImageNet, but more instances per category. It's still shocking to me that they were able to get 2.5 million images labeled at the pixel level. I remember some of the early projects at MIT when crowd-sourced image labeling was just beginning (projects like LabelMe [3]); Antonio Torralba used to joke about how surprised he was about the accuracy of the (nearly) pixel-wise annotations that he was able to crowd-source (and that his mother was a particularly prolific and accurate labeler)!

Instance segmentation turns out to be an very good match for the perception needs we have in manipulation. In the last chapter we found ourselves with a bin full of YCB objects. If we want to pick out only the mustard bottles, and pick them out one at a time, then we can use a deep network to perform an initial instance-level segmentation of the scene, and then use our grasping strategies on only the segmented point cloud. Or if we do need to estimate the pose of an object (e.g. in order to place it in a desired pose), then segmenting the point cloud can also dramatically improve the chances of success with our geometric pose estimation algorithms.

9.1.2 Segmenting new classes via fine tuning

The ImageNet and COCO datasets contain labels for a variety of interesting classes, including cows, elephants, bears, zebras and giraffes. They have a few classes that are more relevant to manipulation (e.g., plates, forks, knives, and spoons), but they don't have a mustard bottle nor a can of potted meat like we have in the YCB dataset. So what are we do to? Must we produce the same image annotation tools and pay for people to label thousands of images for us?

One of the most amazing and magical properties of the deep architectures that have been working so well for instance-level segmentation is their ability to transfer to new tasks ("transfer learning"). A network that was pre-trained on a large dataset like ImageNet or COCO can be fine-tuned with a relatively much smaller amount of labeled data to a new set of classes that are relevant to our particular application. In fact, the architectures are often referred to as having a "backbone" and a "head" -- in order to train a new set of classes, it is often possible to just pop off the existing head and replace it with a new head for the new labels. A relatively small amount of training with a relatively small dataset can still achieve surprisingly robust performance. Moreover, it seems that training initially on the diverse dataset (ImageNet or COCO) is actually important to learn the robust perceptual representations that work for a broad class of perception tasks. Incredible!

This is great news! But we still need some amount of labeled data for our objects of interest. The last few years have seen a number of start-ups based purely on the business model of helping you get your dataset labeled. But thankfully, this isn't our

only option.

9.1.3 Annotation tools for manipulation

Just as projects like LabelMe helped to streamline the process of providing pixel-wise annotations for images downloaded from the web, there are a number of tools that have been developed to streamline the annotation process for robotics. One of the earliest examples was LabelFusion, which combines geometric perception of point clouds with a simple user interface to very rapidly label a large number of images[4].

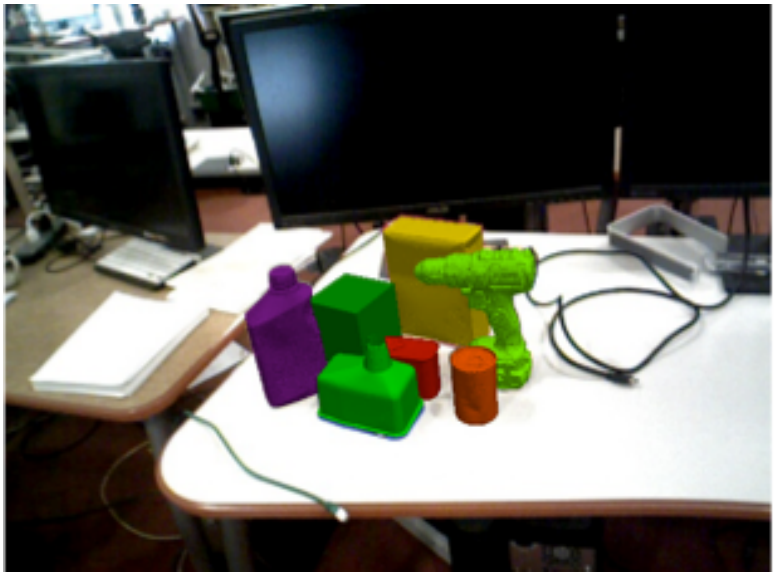


Figure 9.2 - A multi-object scene from LabelFusion [4]. (Mouse over for animation)

In LabelFusion, the user provides multiple RGB-D images of a static scene containing some objects of interest, and the CAD models for those objects. LabelFusion uses a dense reconstruction algorithm, ElasticFusion[5], to merge the point clouds from the individual images into a single dense reconstruction; this is just another instance of the point cloud registration problem. The dense reconstruction algorithm also localizes the camera relative to the point cloud. To localize a particlar object, like the drill in the image above, LabelFusion provides a simple gui that asks the user to click on three points on the model and three points in the scene to establish the "global" correspondence, and then runs ICP to refine the pose estimate. In addition to this one registration providing labeled poses in *all* of the original images, the pixels from the CAD model can be "rendered" on top of all of the images in the established pose giving beautiful pixel-wise labels.

Tools like LabelFusion can be use to label large numbers of images very quickly (three clicks from a user produces ground truth labels in many images).

9.1.4 Synthetic datasets

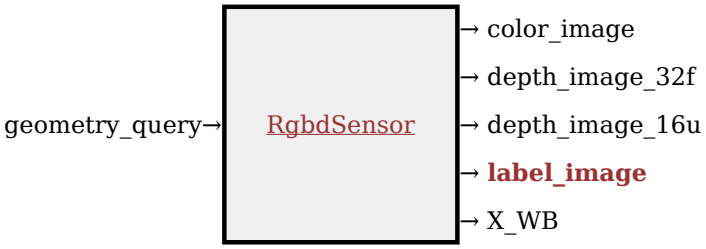
All of this real world data is incredibly valuable. But we have another super powerful tool at our disposal: simulation! Computer vision researchers have traditionally been very skeptical of training perception systems on synthetic images, but as game-engine quality physics-based rendering has become a commodity technology, roboticists have been using it aggressively to supplement or even replace their real-world datasets. The annual robotics conferences now feature regular workshops and/or debates on the topic of "sim2real". For any specific scene or narrow class of objects, we can typically generate accurate enough art assets (with material properties that are often still tuned by an artist) and environment maps / lighting conditions that rendered images can be highly effective in a training dataset. The

bigger question is whether we can generate a *diverse* enough set of data with distributions representative of the real world to train robust feature detectors in the way that we've managed to train with ImageNet. But for many serious robotics groups, synthetic data generation pipelines have significantly augmented or even replaced real-world labeled data.

There is a subtle reason for this. Human annotations on real data, although they can be quite good, are never perfect. Labeling errors can put a ceiling on the total performance achievable by the learning system[6]. Even if we admit the gap between rendered images and natural images, at some point the ability to generate arbitrarily large datasets with perfect pixel-wise labels actually enables training on synthetic datasets to be more surpass the performance for training on real data even when evaluated on real-world test sets.

For the purposes of this chapter, I aim to train an instance-level segmentation system that will work well on our simulated images. For this use case, there is (almost) no debate! Leveraging the pre-trained backbone from COCO, I will use only synthetic data for fine tuning.

You may have noticed it already, but the `RgbdSensor` that we've been using in Drake actually has a "label image" output port that we haven't used yet.



This output port exists precisely to support the perception training use case we have here. It outputs an image that is identical to the RGB image, except that every pixel is "colored" with a unique instance-level identifier.

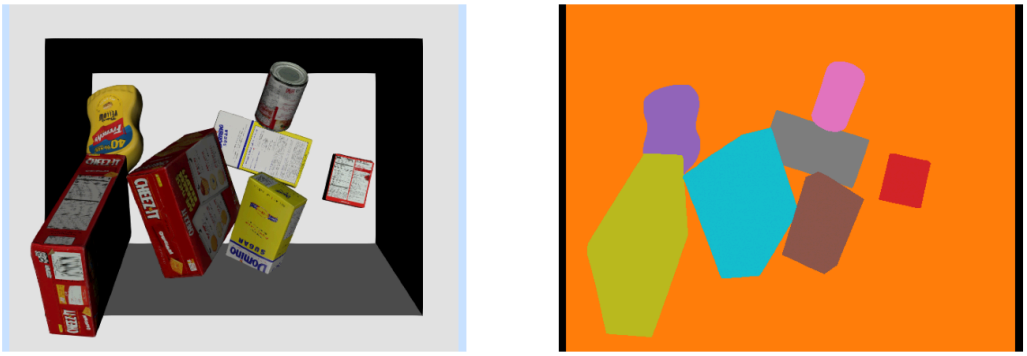


Figure 9.3 - Pixelwise instance segmentation labels provided by the "label image" output port from `RgbdSensor`. I've remapped the colors to be more visually distinct.

Example 9.1 (Generating training data for instance segmentation)

I've provided a simple script that runs our "clutter generator" from our bin picking example that drops random YCB objects into the bin. After a short simulation, I render the RGB image and the label image, and save them (along with some metadata with the instance and class identifiers) to disk.

I've verified that this code *can* run on Colab, but to make a dataset of 10k images using this un-optimized process takes about an hour on my big development desktop. And curating the files is just easier if you run it locally. So I've provided

this one as a python script instead.

```
python3 segmentation_data.py
```

You can also feel free to skip this step! I've uploaded the 10k images that I generated here. We'll download that directly in our training notebook.

9.2 Object detection and segmentation

There is a lot to know about modern object detection and segmentation pipelines. I'll stick to the very basics.

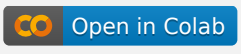
For image recognition (see Figure 1), one can imagine training a standard convolutional network that takes the entire image as an input, and outputs a probability of the image containing a sheep, a dog, etc. In fact, these architectures can even work well for semantic segmentation, where the input is an image and the output is another image; a famous architecture for this is the Fully Convolutional Network (FCN) [7]. But for object detection and instance segmentation, even the number of outputs of the network can change. How do we train a network to output a variable number of detections?

The mainstream approach to this is to first break the input image up into many (let's say on the order of 1000) overlapping regions that might represent interesting sub-images. Then we can run our favorite image recognition and/or segmentation network on each subimage individually, and output a detection for each region that that is scored as having a high probability. In order to output a tight bounding box, the detection networks are also trained to output a "bounding box refinement" that selects a subset of the final region for the bounding box. Originally, these region proposals were done with more traditional image preprocessing algorithms, as in R-CNN (Regions with CNN Features)[8]. But the "Fast" and "Faster" versions of R-CNN replaced even these preprocessing with learned "region proposal networks"[9, 10].

For instance segmentation, we will use the very popular Mask R-CNN network which puts all of these ideas, using region proposal networks and a fully convolutional networks for the object detection and for the masks [11]. In Mask R-CNN, the masks are evaluated in parallel from the object detections, and only the masks corresponding to the most likely detections are actually returned. At the time of this writing, the latest and most performant implementation of Mask R-CNN is available in the Detectron2 project from Facebook AI Research. But that version is not quite as user-friendly and clean as the original version that was released in the PyTorch `torchvision` package; we'll stick to the `torchvision` version for our experiments here.

Example 9.2 (Fine-tuning Mask R-CNN for bin picking)

The following notebook loads our 10k image dataset and a Mask R-CNN network pre-trained on the COCO dataset. It then replaces the head of the pre-trained network with a new head with the right number of outputs for our YCB recognition task, and then runs just a 10 epochs of training with my new dataset.



(Training Notebook)

Training a network this big (it will take about 150MB on disk) is not fast. I strongly recommend hitting play on the cell immediately after the training cell while you are watching it train so that the weights are saved and downloaded even if your Colab session closes. But when you're done, you should have a shiny new network

for instance segmentation of the YCB objects in the bin!

I've provided a second notebook that you can use to load and evaluate the trained model. If you don't want to wait for your own to train, you can examine the one that I've trained!

Open in Colab (Inference Notebook)

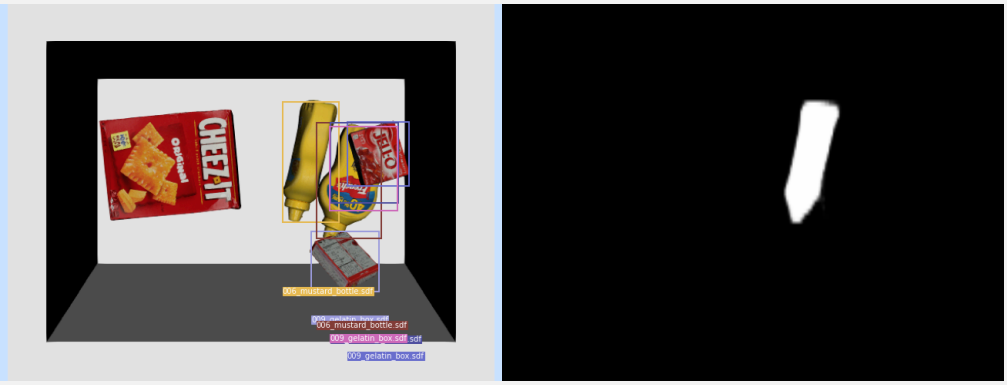


Figure 9.4 - Outputs from the Mask R-CNN inference. (Left) Object detections. (Right) One of the instance masks.

9.3 Putting it all together

We can use our Mask R-CNN inference in a manipulation to do selective picking from the bin...

9.4 Variations and Extensions

9.4.1 Pretraining wth self-supervised learning

9.4.2 Leveraging large-scale models

One of the goals for these notes is to consider "open-world" manipulation -- making a manipulation pipeline that can perform useful tasks in previously unseen environments and with unseen models. How can we possibly provide labeled instances of every object the robot will ever have to manipulate?

The most dramatic examples of open-world reasoning have been coming from the so-called "foundation models"[12]. The foundation model that has been adopted most quickly into robotics research is the large vision + text model, CLIP [13].

More coming soon...

9.5 Exercises

Exercise 9.1 (Label Generation)

For this exericse, you will look into a simple trick to automatically generate training data for Mask-RCNN. You will work exclusively in . You will be asked to complete the following steps:

a. Automatically generate mask labels from pre-processed point clouds.
b. Analyze the applicability of the method for more complex scenes.
c. Apply data augmentation techniques to generate more training data.

Exercise 9.2 (Segmentation + Antipodal Grasping)

For this exercise, you will use Mask-RCNN and our previously developed antipodal grasp strategy to select a grasp given a point cloud. You will work exclusively in . You will be asked to complete the following steps:

a. Automatically filter the point cloud for points that correspond to our intended grasped object
b. Analyze the impact of a multi-camera setup.
c. Consider why filtering the point clouds is a useful step in this grasping pipeline.
d. Discuss how we could improve this grasping pipeline.

References

1. Olga Russakovsky and Jia Deng and Hao Su and Jonathan Krause and Sanjeev Satheesh and Sean Ma and Zhiheng Huang and Andrej Karpathy and Aditya Khosla and Michael Bernstein and others, "Imagenet large scale visual recognition challenge", *International journal of computer vision*, vol. 115, no. 3, pp. 211--252, 2015.

2. Tsung-Yi Lin and Michael Maire and Serge Belongie and James Hays and Pietro Perona and Deva Ramanan and Piotr Doll{\'a}r and C Lawrence Zitnick, "Microsoft coco: Common objects in context", *European conference on computer vision* , pp. 740--755, 2014.

3. Bryan C Russell and Antonio Torralba and Kevin P Murphy and William T Freeman, "LabelMe: a database and web-based tool for image annotation", *International journal of computer vision*, vol. 77, no. 1-3, pp. 157--173, 2008.

4. Pat Marion and Peter R. Florence and Lucas Manuelli and Russ Tedrake, "A Pipeline for Generating Ground Truth Labels for Real {RGBD} Data of Cluttered Scenes", *International Conference on Robotics and Automation (ICRA), Brisbane, Australia*, May, 2018. [link]

5. Thomas Whelan and Renato F Salas-Moreno and Ben Glocker and Andrew J Davison and Stefan Leutenegger, "{ElasticFusion}: Real-time dense {SLAM} and light source estimation", *The International Journal of Robotics Research*, vol. 35, no. 14, pp. 1697--1716, 2016.

6. Curtis G Northcutt and Anish Athalye and Jonas Mueller, "Pervasive label errors in test sets destabilize machine learning benchmarks", *arXiv preprint arXiv:2103.14749*, 2021.

7. Jonathan Long and Evan Shelhamer and Trevor Darrell, "Fully convolutional networks for semantic segmentation", *Proceedings of the IEEE conference on computer vision and pattern recognition* , pp. 3431--3440, 2015.

8. Ross Girshick and Jeff Donahue and Trevor Darrell and Jitendra Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation", *Proceedings of the IEEE conference on computer vision and pattern recognition* , pp. 580--587, 2014.

9. Ross Girshick, "Fast r-cnn", *Proceedings of the IEEE international conference on computer vision* , pp. 1440--1448, 2015.

10. Shaoqing Ren and Kaiming He and Ross Girshick and Jian Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks", *Advances in neural information processing systems* , pp. 91--99, 2015.

11. Kaiming He and Georgia Gkioxari and Piotr Doll{\'a}r and Ross Girshick, "Mask {R-CNN}", *Proceedings of the IEEE international conference on computer vision* , pp. 2961--2969, 2017.

12. Rishi Bommasani and Drew A Hudson and Ehsan Adeli and Russ Altman and Simran Arora and Sydney von Arx and Michael S Bernstein and Jeannette Bohg and Antoine Bosselut and Emma Brunskill and others, "On the opportunities and risks of foundation models", *arXiv preprint arXiv:2108.07258*, 2021.

13. Alec Radford and Jong Wook Kim and Chris Hallacy and Aditya Ramesh and Gabriel Goh and Sandhini Agarwal and Girish Sastry and Amanda Askell and Pamela Mishkin and Jack Clark and others, "Learning transferable visual models from natural language supervision", *International Conference on Machine Learning* , pp. 8748--8763, 2021.

Chapter 10
Deep Perception for Manipulation

In the previous chapter, we discussed deep-learning approaches to object detection and (instance-level) segmentation; these are general-purpose tasks for processing RGB images that are used broadly in computer vision. Detection and segmentation alone can be combined with geometric perception to, for instance, estimate the pose of a known object in just the segmented point cloud instead of the entire scene, or to run our point-cloud grasp selection algorithm only on the segmented point cloud in order to pick up an object of interest.

One of the most amazing features of deep learning for perception is that we can pre-train on a different dataset (like ImageNet or COCO) or even a different task and then fine-tune on our domain-specific dataset or task. But what are the right perception tasks for manipulation? Object detection and segmentation are a great start, but often we want to know more about the scene/objects to manipulate them. That is the topic of this chapter.

There is a potential answer to this question that we will defer to a later chapter: learning end-to-end "visuomotor" policies, sometimes affectionately referred to as "pixels to torques". Here I want us to think first about how we can combine a deep-learning-based perception system with the powerful existing (model-based) tools that we have been building up for planning and control.

I'll start with deep-learning versions of two perception tasks we've already considered: object pose estimation and grasp selection.

10.1 Pose estimation

10.2 Grasp selection

10.3 (Semantic) Keypoints

10.4 Dense Descriptors

10.5 Task-level state

10.6 Other perceptual tasks / representations

My coverage above is necessarily incomplete and the field is moving fast. Here is a quick "shout out" to a few other very relevant ideas.

More coming soon...

10.7 Exercises

Exercise 10.1 (Deep Object Net and Contrastive Loss)

In this problem you will further explore Dense Object Nets, which were introduced in lecture. Dense Object Nets are able to quickly learn consistent pixel-level representations for visual understanding and manipulation. Dense Object Nets are powerful because the representations they predict are applicable to both rigid and non-rigid objects. They can also generalize to new objects in the same class and can be trained with self-supervised learning. For this problem you will work in to first implement the loss function used to train Dense Object Nets, and then predict correspondences between images using a trained Dense Object Net.

Chapter 11
Reinforcement Learning

These days, there is a lot of excitement around reinforcement learning (RL), and a lot of literature available. The scope of what one might consider to be a reinforcement learning algorithm has also broaden significantly. The classic (and now updated) and still best introduction to RL is the book by Sutton and Barto [1] . For a more theoretical view, try [2, 3]. There are some great online courses, as well: Stanford CS234, Berkeley CS285, DeepMind x UCL.

My goal for this chapter is to provide enough of the fundamentals to get us all on the same page, but to focus primarily on the RL ideas (and examples) that are particularly relevant for manipulation. And manipulation is a great playground for RL, due to the need for rich perception, for operating in diverse environments, and potentially with rich contact mechanics. Many of the core applied research results have been motivated by and demonstrated on manipulation examples!

11.1 RL Software

There are now a huge variety of RL toolboxes available online, with widely varying levels of quality and sophistication. But there is one standard that has clearly won out as the default interface with which one should wrap up their simulator in order to connect to a variety of RL algorithms: the Gym.

It's worth taking a minute to appreciate the difference in the OpenAI Gym Environments (`gym.Env`) interface and the Drake System interface; I think it is very telling. My goal (in Drake), is to present you with a rich and beautiful interface to express and optimize dynamical systems, to expose and exploit all possible structure in the governing equations. The goal in Gym is to expose the absolute minimal details, so that it's possible to easily wrap every possible system under the same common interface (it doesn't matter if it's a robot, an Atari game, or even a compiler). Almost by definition, you can wrap any Drake system as a Gym Environment.

Example 11.1 (Using a Drake simulation as an OpenAI Gym Environment)

An OpenAI Gym Environment is an incredibly simple wrapper around simulators which offers a very basic interface, most notably consisting of `reset()`, `step()`, `render()`. The `step()` method returns the current observations and the one-step reward (as well as some additional termination conditions).

You can wrap any Drake simulation in an OpenAI gym environment, using

```
from manipulation.drake_gym import DrakeGymEnv
```

The DrakeGym constructor takes a `Simulator` as well as an input port to associate with the actions, an output port to associate with the observations, etc. For the reward, you can implement it as a simple function of the `Simulator Context`, or as another output port.

`DrakeGym` is built around a `Simulator` (not just a `System`) or a function that produces a random `Simulator` because you might want to control the integrator parameters or have each rollout contain the same robot in a different environment, with potentially different numbers of objects. This would mean that the underlying `System` might have a different number of states / ports. The notion of a function

that can produce simulators, referred to as a `SimulatorFactory`, is core to the stochastic system modeling framework in Drake.

You can also use any Gym environment in the Drake ecosystem; you just won't be able to apply some of the more advanced algorithms that Drake provides. Of course, I think that you should use Drake for your work in RL, too (many people do), because it provides a the rich library of dynamical systems that are rigorously authored and tested, including a great physics engine for dealing with contact, and leaves open the option to put RL approaches head-to-head against more model-based alternatives. I admit I might be a little biased. At any rate, that's the approach we will take in these notes.

Some people might argue that the more thoughtfully you model your system, the more assumptions you have baked in, making yourself susceptible to "sim2real" gaps; but I think that's simply not the case. Thoughtful modeling includes making uncertainty models that can account for as narrow or broad of a class of systems as we aim to explore; good things happen when we can make the uncertainty models themselves *structured*. I think one of the most fundamental challenges waiting for us at the intersection of reinforcement learning and control is a deeper understanding of the class of models that is rich enough to describe the diversity and richness of problems we are exploring in manipulation (e.g. with RGB cameras as inputs) while providing somewhat more structure that we can exploit with stronger algorithms. Importantly, these models should continually expand and improve with data.

The OpenAI Gym provides an interface for RL environments, but doesn't provide the implementation of the actual RL algorithms. There are a large number of popular repositories for the algorithms, too. As of this writing, I would recommend Stable Baselines 3: it provides a very nice and thoughfully-documented set of implementations in PyTorch.

One other class of algorithms that is very relevant to RL but not specifically designed for RL is algorithms for black-box optimization. I quite like Nevergrad, and will also use that here.

11.2 Policy-gradient methods

11.2.1 Black-box optimization

Example 11.2 (Black-box optimization.)
Coming to deepnote soon (I have to handle the dependencies). The example is available in `rl/black_box.ipynb`.

11.2.2 Stochastic optimal control

11.2.3 Using gradients of the policy, but not the environment

You can find more details on the derivation and some basic analysis of these algorithms here.

11.2.4 REINFORCE, PPO, TRPO

Example 11.3 (Flipping up the box.)
Coming to deepnote soon (I have to handle the dependencies). The example is available in `rl/box_flipup.ipynb`.

11.2.5 Control for manipulation should be easy

This is a great time for theoretical RL + controls, with experts from controls embracing new techniques and insights from machine learning, and vice versa. As a simple example, we've increasingly come to understand that, even though the cost landscape for many classical control problems (like the linear quadratic regulator) is not convex in the typical policy parameters, we now understand that gradient descent still works for these problems (there are no local minima), and the class of problems/parameterizations for which we can make statements like this is growing rapidly.

11.3 Value-based methods

11.4 Model-based RL

11.5 Exercises

Exercise 11.1 (Stochastic Optimization)

For this exercise, you will implement a stochastic optimization scheme that does not require exact analytical gradients. You will work exclusively in . You will be asked to complete the following steps:

a. Implement gradient descent with exact analytical gradients.
b. Implement stochastic gradient descent with approximated gradients.
c. Prove that the expected value of the stochastic update does not change with baselines.
d. Implement stochastic gradient descent with baselines.

Exercise 11.2 (REINFORCE)

For this exercise, you will implement the vanilla REINFORCE algorithm on a box pushing task. You will work exclusively in . You will be asked to complete the following steps:

a. Implement the policy loss function.
b. Implement the value loss function.
c. Implement the advantage function.

References

1. Richard S. Sutton and Andrew G. Barto, "Reinforcement Learning: An Introduction", MIT Press , 2018.

2. Alekh Agarwal and Nan Jiang and Sham M. Kakade and Wen Sun, "Reinforcement Learning: Theory and Algorithms", Online Draft , 2020.

3. Csaba Szepesvari, "Algorithms for Reinforcement Learning", Morgan and Claypool Publishers , 2010.

APPENDIX A
Drake

DRAKE is the main software toolbox that we use for this text, which in fact originated in a large part due to the MIT Underactuated Robotics course. The DRAKE website is the main source for information and documentation. The goal of this chapter is to provide any additional information that can help you get up and running with the examples and exercises provided in these notes.

A.1 PYDRAKE

DRAKE is primarily a C++ library, with rigorous coding standards and a maturity level intended to support even professional applications in industry. In order to provide a gentler introduction, and to facilitate rapid prototyping, I've written these notes exclusively in python, using Drake's python bindings (pydrake). These bindings are less mature than the C++ backend; your feedback (and even contributions) are very welcome. It is still improving rapidly.

In particular, while the C++ API documentation is excellent, the autogenerated python docs are still a work in progress. I currently recommend using the C++ documentation to find what you need, then checking the Python documentation only if you need to understand how the class or method is spelled in pydrake.

There are also a number of tutorials in DRAKE that can help you get started.

A.2 ONLINE JUPYTER NOTEBOOKS

I will provide nearly all examples and exercises in the form of a Jupyter Notebook so that we can leverage the fantastic and relatively recent availability of (free) cloud resources.

A.2.1 Running on Deepnote

We'll use Deepnote as the primary platform for the course. After following any of the links from the chapters, you should:

1. Log in (the free account will be sufficient for this class)
2. "Duplicate" the document. Icon is in the top right next to Login.
3. Run all of the cells (can use the "Run notebook" icon just above this cell)
4. Many of the notebooks use MeshCat for interactive visualizations. Click on the url printed just below "StartMeshcat" (often the second code cell of the notebook) to see the MeshCat window.

A.2.2 Running on Google Colab

As of now, Drake no longer supports Google Colab, which is stuck on Ubuntu 18.04 and Python 3.7. We will try to support it again if/when they upgrade.

A.2.3 Enabling licensed solvers

You can enabled more powerful solvers for `MathematicalProgram` if you have a license (most are free for academics). Please see this tutorial for instructions.

A.3 Running on your own machine

As you get more advanced, you will likely want to run (and extend) these examples on your own machine. On platforms that Drake supports (the latest two releases of Mac and Ubuntu), it should be as simple as running

```
pip install manipulation
```

In general, I would strongly recommend running `pip` commands in a virtualenv.

The Drake website also has a number of alternative installation options, including precompiled binaries and Docker instances. If you build from source (awesome!), then make sure you also build the python bindings. Finally, make sure you have the Drake installation in your `PYTHONPATH`.

You'll likely want to start from the manipulation root directory. Then launch your notebook with:

```
jupyter notebook
```

The examples for each chapter that has them will be in a .ipynb file right alongside the chapter's html file, and the notebook exercises are all located in the `exercises` subdirectory.

A.4 Getting help

If you have trouble with Drake, please follow the advice here. If you have trouble with the manipulation repo, you can check for known issues (and potentially file a new one) here.

Appendix B
Setting up your own "Manipulation Station"

Simulation is an extremely powerful tool, and the fact that we can give it away freely makes it a powerful equalizer. Just a few years ago, nobody believed that you could develop a manipulation system in simulation and expect it to work in reality. Nowadays, though, we are seeing consistent evidence that our simulation fidelity is high enough that if can achieve *robust* performance in simulation, then we are optimistic that it can transfer to reality.

Of course, there is no substitute to watching your ideas/code bring life to a real physical robot. The goal of this chapter is to help commoditize the robot hardware infrastructure, to the extent possible, by proving the bill of materials, the low-level drivers, and basic setup instructions to get you up and running with your very own manipulation station.

B.1 Message Passing

Drake supports multiple message passing subsystems (including ROS1), and for the most part I have no strong opinions about what you should use. We use LCM for this class because (1) it is lightweight as a dependency (it would be very painful/ restrictive to have ROS as a mandatory dependency for Drake) and (2) we prefer multicast UDP to tcp/ip for the lowest level communications to/from the robot.

As a result, the primary drivers we list/provide below are based on LCM. I'll try to point to ROS equivalents when possible. If you already have an established ROS workflow and still want to use our drivers, then it is also possible to use simple lcm2ros/ros2lcm "bridge" executables to e.g. publish the LCM messages on a ROS topic (for logging, etc).

B.2 Kuka LBR iiwa + Schunk WSG Gripper

We have made extensive use of both the iiwa7 R800 and the iiwa14 R820; Drake provides URDFs for both models and the drivers should work for either. We use the iiwa7 for the labs in this course because it is slightly smaller and we do not need the end-effector payload.

- IIWA LCM driver

We've been quite happy with the Schunk WSG 50 gripper as a reliable and accurate two-fingered gripper solution.

- Schunk LCM driver

Note: We are evaluating the Franka Panda as an alternative platform (and are likely to provide drivers for it here soon).

B.3 Intel Realsense D415 Depth Cameras

The D415 has been the camera of choice because of it's relatively good minimum range and due to the fact that multiple cameras do not interfere with each other when viewing the same scene.

- LCM Driver
- Camera calibration toolbox.

B.4 Miscellaneous hardware.

For the purposes of having a common workstation that we can use in the labs, we have included here the simple environment in which we mount the robot. The robot just barely fits in the arena -- it was designed with the dimensions limited so that it can be moved (via a manual pallet jack) through standard office doors.

CAD and bill of materials coming soon.

Appendix C
Miscellaneous

C.1 How to cite these notes

Thank you for citing these notes in your work. Please use the following citation:

```
@book{manipulation,
  title        = "Robotic Manipulation",
  subtitle     = "Perception, Planning, and Control",
  howpublished = "Course Notes for MIT 6.4210",
  author       = "Tedrake, Russ",
  year         = 2022,
  url          = "http://manipulation.mit.edu",
}
```

C.2 Annotation tool etiquette

My primary goal for the annotation tool is to host a completely open dialogue on the intellectual content of the text. However, it has turned out to serve an additional purpose: it's a convenient way to point out my miscellaneous typos and grammatical blips. The only problem is that if you highlight a typo, and I fix it 10 minutes later, your highlight will persist forevermore. Ultimately this pollutes the annotation content.

There are two possible solutions:

- you can make a public editorial comment, but must promise to delete that comment once it has been addressed.
- You can join my "editorial" group and post your editorial comments using this group "scope".

Ideally, once I mark a comment as "done", I would appreciate it if you can delete that comment.

I highly value both the discussions and the corrections. Please keep them coming, and thank you!

C.3 Some great final projects

Each semester students put together a final project using the tools from this class. So many of them are fantastic! Here is a small sample that captures some of the diversity.

Fall 2022 Outstanding Project Awards (playlist):

- ChessBot by Ethan Chung
- Rock Skipping Robot by Michael Burgess and Nicholas Ramirez
- How we made a robot arm juggle! by Richard Li, David Jin, and Shao Yuan Chew Chia
- SpeedCuber Bot by Bin Pham
- Michaelangelo - Robot Painter 2.0 by Vainavi Mukkamala and Akila Saravanan
- Ping Pong Bot 2022 by Jenny Zhang and Daniel Klahn
- TetrisBot by Pranav Arunandhi, Ashley Ke, and Sadhana Lolla
- Multiple-finger Grasping: Generating Antipodal Grasping with Allegro Hand

by Yuxiang Ma
- Exploring 6DOF Pose Estimation Approaches Using RGB(D) by Amin Heyrani Nobari

Fall 2021:

- YouTube playlist containing all projects that opted in.
- Playing Piano with a Robotic Hand by Seong Ho Yeon
- Table Cleaning through Task and Motion Planning with Force Control by William Shen
- RallyBot: Exploring Physics-Based Approaches to Robotic Table Tennis by Dylan Zhou and Chaitanya Ravuri
- Robust Stacking of Convex Polytopes by Richard Li and Gabriel Margolis
- Simulation of Discrete Lattice Robot Assemblies by Miani Smith
- Da Vinci Robot by Maisy Lam and Jose A. Muguira
- Certifying Convex Decompositions of Free Configuration Space by Alexandre Amice, Peter Werner, Annan Zhang
- Robotic Arm Weightlifting via Trajectory Optimization by Timur Garipov
- An Efficient Implementation of Pressure Field Models by Vincent Huang, Franklyn Wang, Eric Zhang
- Joint Perception and Manipulation System for Multi-Shape Peg-in-Hole Tasks by Portia Gaitskell and Julia Wyatt

Fall 2020:

- Solving Simple Tiling Puzzles with an End-to-End Robotic System by Anubhav Guha
- Throwing in Drake by Daniel Yang and Tony Wang
- Robot Juggler by Ryan Shubert and Matt Beveridge
- A Geometric Approach to Recycling Cans via Vision-Based Manipulation by Susan Ni and George Chen
- Interactive Perception: Scene Segmentation Through Physical Perturbation by Lukas Lao Beyer and John Keszler
- William Shakebot by Carson Smithbot and Clemente Ocejo

C.4 Please give me feedback!

I'm very interested in your feedback. The annotation tool is one mechanism, but you can also comment directly on the YouTube lectures, or even add issues to the github repo that hosts these course notes.

I will also post a proper survey questionnaire here once the notes/course has existed long enough for that to make sense.