Skills & Knowledge

This section gives a comprehensive list of core robotics principles that every robot software engineer should have awareness of.

This is a big list. *You don't need a deep understanding of every single thing I mention here!* Here's how I would think about it:

- You should have some familiarity with everything mentioned in 'Software Fundamentals' and *deep* familiarity with most of it. Most of the robotics-specific skills require the use of these fundamentals.
- The software fundamentals alone are enough that you could be a useful employee at a robotics company - it will be hard to stand out without robotics-specific knowledge, but practically speaking, most of the actual work you'll do as an early career engineer only requires these things
- You should be familiar with the typical 'system architecture' diagram for robotics and some of the common variations. Be able to explain each component and the data that gets passed between them.
- You should have basic familiarity with the *core concepts* and *intuition* behind most of the individual categories. You don't need to know every algorithm or have a library of projects showcasing these things, but you should be able to talk about what each category is and how it fits into the overall picture
- You should have strong familiarity with at least 2-3 categories. Know the common algorithms and have project-based evidence you can show that demonstrates that knowledge. Be able to go deep on these topics and answer follow-ups.
- In most cases, you won't be expected to be an expert on any of these topics, but if you can demonstrate expertise in a few, it conveys that you *are able to be trained in a specific area*.

If you can achieve all of that, you're in a good position to be hired at a robotics company. For most entry level positions, you won't be expected to go deeper than that in a technical interview. Once you've hit this knowledge threshold, you should be spending much of your time on

- Adding portfolio projects that clearly demonstrate you have this knowledge
- Practicing technical interview questions to make sure you're able to convey the knowledge you have and that you stay sharp on each category
- Optimizing your resume and applications for each job you apply to, to emphasize how your knowledge satisfies what the employer is looking for

At this point, if you have the time, consider going deep on at least one of the topics. If you can say, "I specialize in SLAM / Control Systems / Navigation / …", you will stand out for general purpose roles and also unlock more specialized roles around those topics.

Software Fundamentals:

This section lists general software fundamentals that are pre-requisites for the robotics-specific sections that follow. Aim to be familiar with everything here - if you aren't versed in these topics, learn them first before moving on to the later sections.

Programming Languages

By far the most common programming languages you'll see in robotics are Python and C++. You should be comfortable programming in both of these.

Python:

- Data structures / syntax, functions, error handling, OOP, package / environment management, decorators, common built-in libraries (os, time, logging, json, …), imports / project structure
- Third party packages to know: numpy, scipy, matplotlib, pytest

C++:

- Core C++ principles: header/source, common data types, pointers/references
- OOP: Classes, inheritance, constructors/destructors, operator overloading, abstract classes, pure virtual functions
- Function overloading, default arguments, static
- Templates
- Exceptions, exception handling
- STL:
    - Containers: vector, map, unordered_map, set, deque
    - Algorithms: sort, find, accumulate, transform
- Smart pointers: unique_ptr, shared_ptr
- Iterators, ranges
- File I/O
- <memory>, <thread>, <mutex>, <chrono>, <random>
- External libraries: Eigen, Google Test
- Build: CMake (know enough to get by - deeper knowledge a big plus though)
- Header guards / pragma, organizing libraries / executables in CMake, static vs shared libraries
- Debugging: GDB basics

Going deeper: pick any of these categories and become an 'expert', some especially useful focus areas: multithreading, memory management / shared memory, debugging, build process.

Learning Resources:

- YouTube, ChatGPT + doing lots of projects that use these different things

Exercises:

Just make sure you're using Python or C++ for the exercises in the other sections!

Interview Questions:

Python:

- What are the differences between lists, tuples, sets, and dictionaries? When would you use each?
- Explain the difference between `is` and `==` in Python.
- What is the difference between shallow copy and deep copy? How do you create each?
- What is the purpose of `*args` and `**kwargs` in function definitions?
- How are Python variables scoped (LEGB rule)?
- What's the difference between a class method, an instance method, and a static method?
- How does Python implement inheritance and method overriding?
- What are `__str__` and `__repr__` used for?
- How do properties (`@property`) differ from getter/setter methods?
- How does Python handle exceptions? Give an example with `try`/`except`/`finally`.
- What's the difference between catching a base `Exception` vs a specific exception type?
- How would you read/write a text file in Python?
- What's the difference between using `os.path` and `pathlib`?
- How do you log information properly in Python (instead of `print`)?
- Explain the difference between threading, multiprocessing, and asyncio in Python.
- Why does the Global Interpreter Lock (GIL) matter? When is it a problem?
- How would you speed up a loop in Python without changing the algorithm?
- How would you write a simple test case with `pytest`?
- What's the role of a `requirements.txt` or `pyproject.toml` file?

C++

- What's the difference between a pointer and a reference?
- What's the difference between `struct` and `class` in C++?
- What does RAII mean and why is it important?
- Explain the rule of 5 (copy constructor, copy assignment, move constructor, move assignment, destructor).
- What is the difference between `const int*`, `int* const`, and `const int* const`?
- When would you use `new`/`delete` vs smart pointers (`std::unique_ptr`, `std::shared_ptr`)?
- What's the difference between stack allocation and heap allocation?
- How do `std::move` and `std::forward` work?
- What's the difference between `virtual`, `override`, and `final`?
- What is a pure virtual function?
- When would you make a destructor `virtual`?

- What is function overloading vs operator overloading?
- What are the differences between `std::vector`, `std::list`, and `std::deque`?
- How does `std::map` differ from `std::unordered_map`?
- Show how you would time a function execution using `<chrono>`.
- What is a function template? Give an example.
- Explain the difference between compile-time polymorphism (templates) and runtime polymorphism (virtual functions).
- What are C++17/20 features like `std::optional`, `std::variant`, or concepts useful for?
- How do you create a thread in C++ using `<thread>`?
- What's the difference between a mutex and a condition variable?
- What is a race condition? How would you prevent one?
- What tools would you use to debug a memory leak in C++?
- What does Valgrind or AddressSanitizer do?

Mixed / Comparative

- When would you choose Python vs C++ for a robotics project?
- How does memory management differ between Python and C++?
- Show the same algorithm (e.g., summing an array) in both Python and C++ — what are the trade-offs?
- How would you bind a C++ library into Python (e.g., `pybind11`)?

Linux

You should know your way around a Linux system and be familiar with the most common commands. You won't be expected to be an expert or know every command, but the ability to work on / deploy software to a Linux-based system is a must-have.

- Commands: this list is a good place to start. If you're familiar with most of these you'll be in good shape
- Filesystem: know your way around the filesystem of any common Linux distro. Know what each of the root level directories is generally used for
- Package management - know how to install packages and how package management works
- SSH, SCP
- Systemctl, scheduling services to run on startup, setting startup order

Exercises:

- Use Linux for all the other exercises in this guide, unless there's a specific reason not to
- Make flashcards for the 100 most common commands and commit all of them to memory
    - Memorize the *explanation* of what they do
    - Memorize the *usage* - try to use the command without looking anything up
    - Try to learn 3-5 new commands per day. Any time you encounter a new command, add it to your deck

    - If you do this for a month, you'll be in great shape

Interview Questions:

- What Linux distributions have you used / what's your comfort level with Linux?
- How do you check the kernel version and system resources (CPU, memory, disk)?
- Explain the difference between a process and a thread. How would you see them in Linux?
- What's the difference between `/dev`, `/proc`, and `/sys`?
- How do you find a process that's consuming too much CPU?
- What command would you use to check which processes are using a specific port?
- How do you search for a string inside files recursively?
- How would you copy a folder from one machine to another over SSH?
- How do you make a script executable?
- How do you check disk usage on the system?
- What's the difference between relative and absolute paths in Linux?
- Explain Linux file permissions (r, w, x) and how to change them
- How would you mount a USB drive?
- How do you check your IP address and network interfaces?
- What's the difference between `ping`, `traceroute`, and `netstat`/`ss`?
- How do you start/stop/restart a service (e.g., `systemctl`)?
- How do you kill a process by name or PID?
- What's the difference between foreground and background jobs in Linux?
- How would you schedule a task to run periodically?
- How do you compile C++ code on Linux without an IDE?
- What's the difference between `make` and `cmake`?
- How do you check which libraries a binary depends on?
- How do you read log files in real time?

Git & Version Control

Most robotics companies will use Git for version control. You don't need to be an expert, but you need to know enough that you won't screw things up.

- Git basics: add, commit, push, pull, merge
- Common branching strategies for larger/team software projects
- How pull requests work, why they're useful
- Be familiar with GitHub or another similar service
- Know how to:
    - Create / clone a repo
    - Create / switch branches
    - Reverting changes
    - Dealing with merge conflicts

- Know Git command line basics (enough to be able to do all the things listed above)

Exercises:

- For any larger projects you work on (1+ weeks), pick one of the common branching strategies and use it for that project. It might be overkill but it will get you comfortable with the commands and workflow

Interview Questions:

- What's the difference between Git and GitHub/GitLab/Bitbucket?
- What is the difference between a local repository and a remote one?
- How do you clone a repository?
- How do you create a new branch? Why use branches instead of committing to `main`?
- What's the difference between `git pull` and `git fetch`?
- How do you stage and commit changes?
- How do you view commit history?
- What's the difference between `git merge` and `git rebase`? When might you use each?
- How would you resolve a merge conflict?
- How do you check what branch you're currently on?
- How do you undo the last commit but keep the changes in your working directory?
- How do you discard changes in a single file?
- What's the difference between `git reset` and `git revert`?
- What is a pull request and why might you require them for a project?
- What's the difference between forking and branching?
- How do you check who last modified a particular line of code?
- Explain the difference between `HEAD`, `working directory`, and `index/staging area`.
- What does a "detached HEAD state" mean? How do you get out of it?
- How do submodules work, and why might you use one?
- What's Git LFS (Large File Storage), and when is it needed?

Debugging / Logging / Testing

Different companies will have different practices around each of these, but make sure you know the fundamentals within each category. Be able to talk about the importance of each thing, and have at least some experience with doing them in practice.

- Debugging
    - Basic stepping with GDB / LLDB: breakpoints, run, step/next, frame, print
    - Coredumps: how to capture, how to inspect later
    - Compiler / build options for enabling warnings
- Logging

    - Logging levels: DEBUG/INFO/WARN/ERROR/FATAL
    - Python ‘logging’ library
- Testing:
    - Unit tests vs. integration tests vs. system tests
    - Writing tests with GoogleTest/GoogleMock, Pytest
    - Be able to come up with a set of tests to adequately cover a given function
    - Structure of a good test - Arrange-Act-Assert

Exercises:

- Practice using GDB to debug your C++ code as you work through the various other exercises here. Get comfortable with the basic commands and work on debugging faster and more efficiently
- For any complicated projects (1+ weeks), especially those with multiple files or subsystems, use structured/leveled logging. Make sure when something goes wrong, there’s logging in place that makes it clear where the problem happened

Interview Questions:

- How do you find memory leaks in a C++ application?
- What flags would you use to compile for debugging vs release?
- Explain how you would use `gdb` to inspect the call stack of a crashing process.
- How would you debug a Python script that hangs without throwing an error?
- What’s the difference between using `print` statements and using `pdb`?
- A Python process slowly grows in memory usage — how do you figure out if it’s a memory leak?
- What’s the difference between `INFO`, `DEBUG`, `WARNING`, `ERROR`, and `FATAL` log levels? When would you use each?
- If your logs are overwhelming (thousands per second), how would you reduce or manage them?
- Show how you’d configure Python’s logging module to log to both console and file with timestamps.
- What framework would you use for C++ unit testing? For Python?
- What’s the difference between a mock and a stub?
- How would you set up tests so they run automatically on every pull request?
- What’s the role of code coverage tools? How do you use them in C++ vs Python?

Algorithms & Data Structures

- Core data structures: arrays, linked lists, stacks, queues, hash tables / maps, trees, graphs, heaps / priority queues
    - Don’t need to know how to implement these from scratch - more important is knowing use cases for each, lookup efficiency, tradeoffs, etc.
- Algorithms: familiarity with a variety of general purpose algorithms (sorting, searching, etc) is good but don’t worry about memorizing every little thing. For the most part, I recommend focusing on the robotics-specific topics covered later in this guide

- Complexity - be able to do basic Big-O analysis on a simple algorithm and demonstrate that time and space complexity are something you think about when writing an algorithm
- In particular, know how hash tables can be used to speed up various algorithms - I find that this comes up surprisingly often in interviews

Exercises:

- Do 1 or 2 LeetCode style practice problems per day to stay fresh and get comfortable solving this kind of problem quickly. No need to go overboard with this though.
- Similarly, read up on one or two common algorithms each day and ideally implement it (try and do this from scratch without having ChatGPT do it for you).

Interview Questions:

One or two live coding questions is fairly common for interviews. These generally aren't super difficult and are meant to test basic coding competence and ability to break down a problem. If you do a daily practice problem, you should handle this pretty easily.

- How would you reverse a string in-place? What's the complexity?
- You have a large array of sensor readings — how would you efficiently find the maximum?
- What are the trade-offs of using a linked list vs an array?
- How would you detect if a linked list has a cycle?
- Give an example of when you'd use a stack vs a queue.
- How would you implement a queue using two stacks?
- Why are hash maps useful for storing configuration parameters?
- What happens when many keys hash to the same bucket?
- What's the difference between a binary tree, binary search tree, and heap?
- How would you do an in-order traversal of a binary tree?
- How might you represent a graph in code?
- What's the difference between BFS and DFS? When would you use each?
- How do you implement a min-heap using an array?
- What is the complexity of quicksort in best/worst cases?
- Why might mergesort be preferable for linked lists?
- What are the requirements to use binary search?
- How would you search for a target value in a sorted but rotated array?
- Can you explain memoization vs tabulation?
- Solve the Fibonacci sequence with DP.
- What is a greedy algorithm

Networking and Data Transfer

Robotics typically involves both separate processes within a computer talking to each other, as well as communication between different robots, base stations, and cloud servers. While you don't need to be a networking expert, you'll be expected to know (and will frequently) use certain fundamentals:

- Internet protocol basics: IP addresses, ports, hostnames
- Sockets
    - What they are, how they work
    - TCP/UDP: what's the difference, when would you use one vs. the other?
    - How to create one in both Python and C++
- How to serialize data to send across the network: JSON, Protobuf
- HTTP & APIs:
    - Be able to explain how HTTP works, how it relates to TCP/IP
    - Know what an API is and the common ways that they're organized
        - GET, POST, PUT, PATCH, and DELETE requests

Exercises:

1. Write two different programs, running as separate processes on the *same* system. Implement a simple client/server on each side using TCP or UDP. Make it so when you input text into one program, the other program prints it out, and vice-versa.
    a. Extend the programs from the prior example to work between two different computers (on your home network)
    b. Re-write one of the programs so that one side is written in Python and the other is in C++
    c. Modify the programs so that you can pass *structured* data between them, rather than plain text. Use JSON-formatted messages or Protobuf
2. Find an interesting web API (e.g. OpenAI chat interface, DeepL text translation, …). Practice sending requests to that API and parsing the responses. Try with Curl, a Python script, and with Postman or another similar tool
3. Implement a simple API using a python package of your choice. Expose a few different endpoints for different request types. Write a separate program that hits the different endpoints of your API.

Interview Questions:

- What's the difference between TCP and UDP? When would you use one over the other in robotics?
- How does a three-way handshake work in TCP?
- What are common causes of packet loss? How can you detect and mitigate it?
- Explain the difference between latency, throughput, and jitter. Which matters most for real-time robot control?
- How do IP addresses, ports, and sockets fit together?
- Write or explain how you'd create a simple TCP socket server in Python or C++.
- How would you make a non-blocking socket? Why might this be important?
- What's the difference between blocking, non-blocking, and asynchronous I/O?

- Suppose two devices are exchanging messages over UDP — how do you ensure messages arrive in order?
- How would you debug if a socket connection works on localhost but fails across the network?
- Why might a LiDAR driver use UDP instead of TCP?
- How does HTTP differ from raw sockets? Why might you use an HTTP API for a cloud service but not for real-time control?
- What's the difference between REST APIs and WebSockets?
- How do you authenticate when calling an API (tokens, API keys, OAuth)?
- How would you check if a port is open and listening?
- Which tools would you use to capture and inspect packets?
- A robot publishes sensor data over UDP, but the base station isn't receiving it — what steps would you take?

Other Nice-To-Have Expertise:

Docker

- What it is, why it's useful
- How to write a simple Dockerfile
- Images vs containers
- How to build and run a docker container
- Volumes
- Networking
- **Exercise**: choose any program you write for a different exercise - containerize it with docker, create an image, transfer the image to another system, and run the container on that system

DevOps, CI/CD basics

- Know what CI/CD are, understand what a typical pipeline looks like
- Know some of the common tools - GitHub Actions, Jenkins, etc.
- **Exercise**: set up a simple build → test → deploy pipeline for one of your projects

Multithreading

- Difference between processes and threads
- Communication between threads - shared memory, message-passing
- Common issues: race conditions, deadlocks
- Thread synchronization: mutexes, locks, condition variables
- Significance of Python GIL for threading
- C++: std::thread, std::mutex, std::async

Cloud services platform

You generally won't need to work with one of these directly, so this is very much a 'nice-to-have', but most robotics companies will use cloud compute in some way - fleet management, data collection, enterprise layer - maybe even off-loading heavy compute operations from robots in some cases. So it will be helpful to have some familiarity with one of the main cloud services platforms (GCP, AWS, Azure)

- Compute: EC2, Compute Engine, …
- Storage s3, GCS, …
- Databases: SQL, NoSQL, Postgres, …

System Architecture

A robotics software stack can generally be viewed as a set of connected *subsystems*, where each subsystem has a focused role and is characterized by its inputs and outputs - what data (or action) does the subsystem produce, and what data (or stimulus) does it require?

These are some subsystems that appear across many types of robots:

Localization
Inputs → world map, sensor data, movement intent
Output → Estimate of the robot's position in the world

Path Planning
Inputs → world costmap, current position, target position
Output → lowest-cost path from the current position to the target position

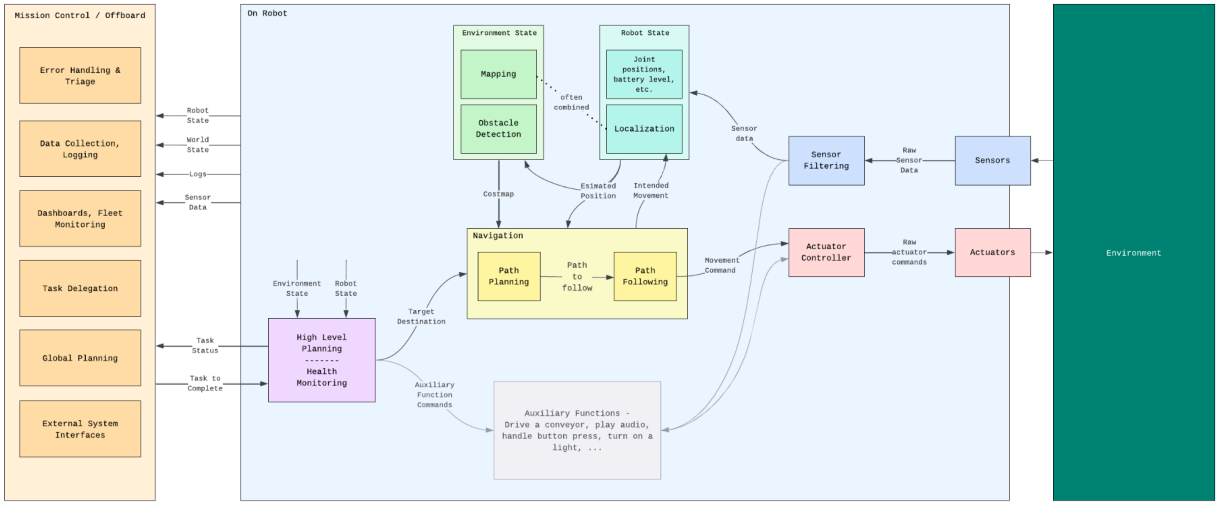
Obstacle Detection
Inputs → sensor data (camera, LiDAR), current position
Output → local occupancy map / costmap indicating safe vs. unsafe terrain

As an early-career robot software developer, you won't be expected to be an expert on each subsystem. As a robot and its environment become more complex, each of these domains often grows into a very challenging problem, requiring an entire team dedicated to solving it. Many robotics engineers will specialize in one discipline (e.g. localization, control, perception) - developing a deep familiarity with one of these problem-sets is a good way to improve your employability.

However, it's always important to understand the overall picture of how the pieces of software fit together, what each piece does and what data is passed between them. Understanding the big picture:

- Shows you how the piece you're working on fits into the overall system
- Helps you foresee and avoid integration issues by clarifying how each system affects the others
- Unlocks roles focused on system design / architecture and makes you better able to communicate with people in those roles
- Makes it easier to hop around between different parts of the system. This is especially important early in your career when your main goal is to learn and gain exposure to lots of different problems.

The following diagram shows the basic system architecture and data flow for a generic mobile robot. For an entry level role, you should be able to explain what each box is doing, and how all the pieces connect to create a functioning robot. Sections later in this guide will describe each subsystem in greater detail.



This diagram is a very high-level overview for one type of robotics application. Once the pieces here make sense to you, you should start to think about how we might modify it…

- What information is missing? Where can we add more details (specific algorithms, data types, additional boxes, etc)?
- What would this diagram look like for a robot arm? How about for a satellite? What would change if we're designing an aquatic robot vs. a land robot?

From diagrams to Code

Analyzing robot software architecture visually is especially useful because the software is often implemented as a set of discrete modules (also called nodes, services, or subsystems) that map almost one-to-one to the boxes in this type of diagram.

Each subsystem can be thought of in terms of *what data it produces* and *what data it requires*. As a developer, the hard work is in figuring out exactly how to turn those inputs into a trustworthy set of outputs. Much of the rest of this guide will be focused on detailing the common approaches to implementing these subsystems.

Before we dive into those details, there are two key questions we need to be able to answer that apply to all subsystems.

1. How is the subsystem structured - how do we make it run continuously, define its loop rate, and determine if it's 'healthy'?
2. How does it talk to the other subsystems?

State machines

It’s very common for robotics subsystems to be implemented as *state machines*. Make sure you understand what a state machine is, why they’re useful, and how to implement one. I’ll review some of the key ideas here and provide other resources for a deeper explanation.

A state machine (sometimes ‘finite state machine) is a piece of software that has a discrete set of pre-defined *states*. At any point in time, one state is active. The active state only changes when a particular condition is met, causing it to *transition* to a different state.

As an example, a *Path Planning Subsystem* might have these states:

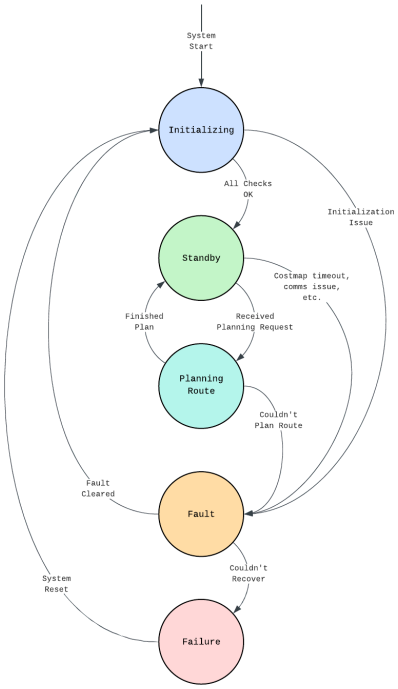
INITIALIZING: The default state on startup - initialize any variables, wait for costmap to be received from the state estimation subsystem

STANDBY: Subsystem is ready to plan routes, but has not received any request (or has finished any previous requests)

PLANNING_ROUTE: High-level planning subsystem has sent a new target destination. Path planner is running the path planning algorithm to determine the shortest path.

FAULT: A recoverable error has occurred. Failed to plan a route, received a target destination that’s outside the map, etc.

FAILURE: An un-recoverable error has occurred



- - - - - - - - - -

State machines ensure that the software state of each subsystem is always legible and clearly defined. The explicitly defined state transitions make it easy to determine what error occurred, why it occurred, and to trace cascading failures affecting multiple subsystems back to their source. They also simplify higher level health monitoring - at any moment, we can see the state of each component of the system and detect if anything looks abnormal.

In robotics, we often use state machines with a fixed *loop rate*. On every iteration, the current state runs, and any transition is applied, if one occurred. The subsystem then waits for the loop time to elapse before running the next state. This lets us allocate compute resources based on the timing requirements of each subsystem. The motor controller can run very fast (100 Hz or more), whereas 2 Hz might be sufficient for the high-level planner which only receives occasional tasks from mission control.

Communication strategies

With software divided into discrete modules, we need a robust way for each subsystem to communicate with the others, and for the robot to communicate with ‘mission control’ and/or other robots. This involves some combination of:

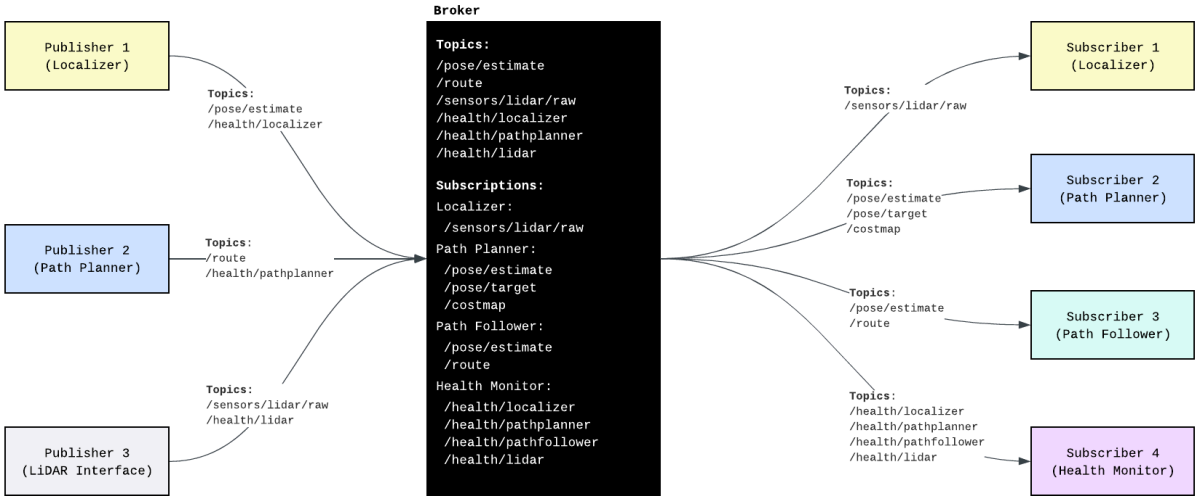
- One-to-one communication - e.g. mission control sending a new task to a robot
- One-to-many communication - e.g. localization module sending the position estimate to many other subsystems
- Many-to-one communication - e.g. the health monitoring system needs information from each of the other subsystems

Pub/Sub Framework

One of the most common ways to handle this communication is with a publisher / subscriber communication model. Make sure you’re familiar with what this is and what problems it solves within a robot software stack.

I’ll provide resources to go deeper on a few pub/sub frameworks, but the core concepts you should understand are:

- Topic: a channel that carries one type of message. This is what publishers/subscribers publish/subscribe to
- Message: a discrete packet of information, e.g. robot pose, sensor reading, motor control command
- Publisher: A piece of software that sends messages on a particular topic
- Subscriber: A piece of software that should receive messages published on a specific topic
- Broker: The piece of software that coordinates all of the communication. It keeps track of what topics are in use, who is publishing to each topic, who is subscribed to each topic, and routes the messages published on each topic to each of the subscribers



The exact terminology for each of these components may vary between frameworks, but the general model will be the same. This model works well because it enables all the communication types mentioned above, and easily allows for new modules to be added / swapped / removed without needing to re-write any of the messaging infrastructure.

The most widely used pub/sub framework for robotics is **ROS2** (Robot Operating System). ROS2 provides much more than just a pub/sub framework but that is one of its core components. I'll cover ROS a bit more later, but here I'll just recommend reading some of the documentation about setting up a pub/sub network with ROS and ideally working through some simple projects on your own to get comfortable with the core concepts.

Another popular tool is MQTT - MQTT implements the pub/sub messaging model but doesn't include all the other tools that ROS does, which could be good or bad depending on your use case.

From a job-interview perspective, deep knowledge isn't essential here, but it's a good way to stand out. Many robotics companies will be using one of these frameworks (or a similar one) and seeing that you are already familiar with it and won't have to learn it on the job is a big plus.

Other communication strategies

A pub/sub framework can generally handle most of the inter-subsystem communication, but there are a few other approaches that you may see for specific use cases. I recommend reading up on each of these and having a basic understanding of how each one works and when it might be useful:

- HTTP API: good if you need to serve a control panel or dashboard, or for managing configs or other resources where GET / POST / DELETE style interface makes sense
- Direct TCP/UDP sockets: good if you have very low latency requirements or need more flexibility that the pre-build frameworks allow

- Shared memory: useful if you need to share large data between processes running on the same device. Can be much faster than socket-based communication

Exercises:

1. Create a system architecture diagram similar to the one in this section for
    a. An industrial robot arm
    b. A mars rover
    c. A pizza delivery drone
    d. A rocket
2. Pick a few subsystems from the examples in exercise 1. Sketch out a state machine diagram for that subsystem - include both the states and the possible transitions between the states.
3. Think through some of the possible failure modes for each subsystem. What downstream effects would this failure have on other subsystems?
4. Implement a state machine in Python or C++ for a system of your choosing
    a. Start by choosing a (simple) system and describing how it works
    b. Drawing out each state and each transition condition
    c. Implement the state machine in Python or C++
    d. Add logs so you can see when each transition occurs and which condition triggered each transition
5. Pick either ROS2 or MQTT (or both!), create a single publisher and subscriber. The publisher should publish text that you input from the command line, and the subscriber should print out the text.
6. Combine your state machine and your pub/sub network. Modify your state machine so that one or more state transitions is triggered by a message received on the pub sub network. Have your state machine publish a ‘health’ update once per second that indicates the current state, plus any other information you want to include.

Interview Questions:

- You’ve been put in charge of designing the software for a completely new robot [think of the first robot that comes to mind and use that]. Before you write any software, what things are you thinking about? How do you break down the problem? What infrastructure do you want to put in place / what do you want to decide on before you write code?
- What is a finite state machine and why are they useful in robotics?
- What are some alternatives to state machines for implementing the software components of a robot? When might you prefer to use one of the alternatives?
- Why might different subsystems run at different loop rates? Give examples.
- How would you detect if a subsystem has stopped functioning correctly?
- Describe the pub/sub communication model. Why is it used in robotics?
- Explain the basics of a pub/sub framework that you’re familiar with.
- What are some situations where a pub/sub communication model doesn’t make sense? What would you use instead?

- What is shared memory? Why is it useful? What are its drawbacks?

Fundamental Robotics Concepts:

In this section, we'll go one-by-one through a dozen or so topics that are likely to be important in most robotics applications.

There is a *lot* of information here and I don't want you to be overwhelmed by it. *You do not need to know everything here!* Your priorities should be:

1. Have *basic familiarity* with most of the topics covered here - enough that you can talk intelligently about them, you understand how they fit into the overall picture, and that if called upon, you could dig deeper into that topic and actually implement code.
2. Have deep knowledge of *one or two* topics. Implement several of the algorithms in that category and consider working on a more complex project to showcase that skill in your portfolio.

This level of coverage is sufficient for early-career job-seekers because it shows that you have broad knowledge of many robotics topics, and are *capable* of deep-diving narrow topics when called upon. This is generally what employers want to see - if we assign you to a specific problem, do you have enough background knowledge to get started, and are you capable of going deep enough to really own that problem and develop a solution.

For each topic, I've included the following sections

- **Use cases / motivation:** why this topic is important, what problems it tries to solve, and specific examples of where it applies
- **Key ideas:** the core intuition, techniques, and algorithms you should be aware of. I'll include a brief overview and links to resources to learn more
- **Exercises**: ideas for hands-on ways to get comfortable with the topic and related algorithms
- **Interview questions**: sample technical questions you might be asked that relate to the topic

Each topic goes *much* deeper than what I've included here. My goal is to include the key ideas that an early-career developer should be aware of, plus some resources for going deeper, but these sections are not comprehensive. My intent is that if you're comfortable with all the topics I cover, you'll be in great shape to land a job in robotics.

You can specialize in just about any of these sections - if there's one that seems especially interesting, you should absolutely lean into that interest. Becoming a true expert in any of the topics here will open up a whole new category of job opportunities, and will make you stand out even more in non-specialized roles.

Geometry & Kinematics:

Use Cases / Motivation

The topics in this section should help us answer the following questions:

- How do we represent the position of our robot in the world?
- How do we take sensor measurements that are relative to the sensor, and get their position relative to
- some other part of the robot or relative to the world?
- If we know the angle of each of our robot's joints, how can we figure out the pose of the end-effector? Or, if we know where we want the end effector to be, how do we set each joint to achieve that pose?
- If our robot is following a path, how can we determine its progress relative along that path?

Key Ideas

Coordinate frames and transforms

Things to know:

- *Coordinate frames* - what they are, how you represent an object's position in a frame, examples of frames that are frequently used in robotics [1]
- [Math] *Basis vectors*, how to perform '*change of basis*' on a set of points [1]
- *Rigid-body transforms* - *translation* vs. *rotation*, how to apply each one using a matrix / vector [1, 2]
- *Homogeneous coordinates* - know how to apply rotation and translation with a single matrix multiplication [1, 2]
- How to *chain transforms* together
- How to apply *inverse transforms*

General Links: [1]

Euler angles and quaternions

Things to know:

- *Euler angles vs. Quaternions* - how to use each to represent orientation, advantages and disadvantages of each one
- Conversion between the two representations
- Conversion between each representation and a rotation matrix
- How to *interpolate* between two orientations and get smooth, shortest-path transitions

Forward kinematics

Things to know:

- Terminology - *kinematic chain*, *revolute joints*, *prismatic joints*, *links* / *link frames*, *transform chain*
- *Denavit-Hartenberg Convention* for assigning coordinate frames to links
- Given a set of joint parameters, be able to compute the pose of the end-effector

Inverse kinematics

Things to know:

- Why IK is challenging - *non-uniqueness*, *unreachable poses*
- *Analytical solutions* - when they're applicable and how to derive them
    - Be able to derive one for a basic 2 or 3 joint robot arm and handle the multiple / no solution problems
- Numerical solutions - *Newton-Raphson*, *Jacobian inverse*, *gradient descent*
- *Jacobian matrix* - what it is / what it does, why it's useful in IK solvers, how it helps with detecting singularities
- *Singularities* - what they are, common examples in robot arms, how to detect them and avoid them

More linear algebra - dot and cross product, projections, basis changes, norms

Things to know:

- Dot product - how to compute and how to interpret the result
- Computing scalar projection and vector projection of one vector on to another
- Cross product - how to compute, geometric interpretation
- Robotics applications of these concepts:
    - Computing angle between robot heading and target direction
    - Determining robot's distance travelled along a path
    - Breaking velocity down into components along / orthogonal to a path
    - Finding an orientation for an end-effector that's perpendicular to a surface
- Norms - how to compute L1 and L2 norms

Tools / Software

- Eigen (C++) - know how to add it to your project, construct vectors and matrices, and implement some of the math covered in this section
- Numpy + Scipy (Python) - Likewise, know how to implement the math covered here using these libraries

- ROS tf2 (C++ or Python) - know how to use tf2 to construct a transform tree and convert between different frames

Exercises

1. Draw a diagram of a simple mobile robot - mark the center, and the relative positions of a camera and wheels.
    a. Give the center of the robot a random position / orientation in the world frame. Compute the positions of the wheels and camera in the world frame. Do this by hand first, then write a program to do it
    b. Modify your program so it uses tf2 to keep track of the transforms
    c. Make the program interactive - plot the position of each frame in a window, and make it so you can move the robot using the arrow keys and the position of each frame updates.
2. Write functions that perform each of the following (avoid ChatGPT or built-in functions, goal here is to gain intuition)
    a. Dot product
    b. Angle between two vectors
    c. Cross product
    d. Vector projection
    e. Scalar projection
    f. Euler > Quaternion conversion, Quaternion > Euler conversion
    g. Euler/Quaternion > Rotation matrix conversion, Rotation matrix > Euler/Quaternion conversion
3. Make a minimal simulation of a 2-segment robot arm. Give each segment a hard-coded length and track the position and angle of each joint in space
    a. Use matplotlib (or your preferred visualization tool) to visualize the joint and segment positions
    b. Add an input that lets you adjust the angle of each joint. When you change an angle, update the visualization so all of the joints are shown at the correct positions
    c. Add an input that lets you set the end-effector position. When you set the position, compute the joint positions needed to put the end-effector in that position and update the visualization
    d. Add interactive sliders that let you control the arm (in both ‘forward’ and ‘inverse’ modes)
    e. Make your arm robust to common issues like multiple solutions, unreachable poses, and singularities

Interview Questions

- Given a point in a robot’s base frame, how would you express it in the world/map frame?
- What’s the difference between a rotation matrix and a homogeneous transform?

- How would you manage / keep track of a large number of reference frames, so that it's easy to express the position of any frame relative to any other one?
- What are the advantages and disadvantages of Euler angles versus quaternions?
- What is gimbal lock? Can you give an example of when it occurs?
- How do you normalize a quaternion, and why is this necessary?
- Suppose you're given two 3D vectors. How can you find the axis and angle that rotates one into the other?
- Explain the difference between forward and inverse kinematics, and give examples of when you would use each? (give a robot arm example and a *non*-robot arm example for each)
- What are Denavit–Hartenberg parameters, and why are they useful?
- What are some problems you might run into with an IK solver and how can you handle them?
- Give two approaches for solving IK if a closed-form solution doesn't exist
- Suppose your IK solver doesn't converge. What steps might you take to debug it?
- What does the Jacobian represent in robot kinematics?
- How can you detect if a manipulator is at a singularity?
- Your car is merging onto a straight highway. You know the speed of the car in the merge lane as well as the merge angle. How do you determine (a) how fast the car is moving in the direction of the highway lanes, (b) how fast the car is moving in the merge direction?
- Given two quaternions representing orientations, how would you compute the "difference" between them?
- Your drone occasionally flips uncontrollably in simulation when making sharp turns. What might be happening?
- If you only had 30 minutes to test whether your IK implementation works, what quick tests would you run?

State Estimation

Use Cases / Motivation

The topics in this section should help us answer the following questions:

- How do we obtain a smooth estimate of a quadcopter's attitude, when our sensors have noise and drift?
- How do we properly combine information from different types of sensors, e.g. IMUs, encoders, and GPS?
- How do we know how confident we should be in our current position estimate, given the limitations of our sensors?

Key Ideas

Uncertainty, noise, bias, drift

Things to know:

- Terminology: Noise (measurement noise, process noise), bias, drift, mean, variance, covariance, expected value, signal-to-noise ratio, prior vs posterior
- Gaussian distribution - know what this distribution is, its key parameters, how it can be used to model noise, and what it's limitations are
- How uncertainty evolves over time - growing in the absence of measurements, shrinking as measurements are made
- How sensor calibration helps us better understand and manage uncertainty
- Dead reckoning - what it is, when it's used, why it causes problems
- Absolute measurements vs relative measurements

Sensor fusion

Things to know:

- Core idea - combining sensors with different strengths / weaknesses to overcome their individual limitations
- Common sensors and what their limitations are: wheel/motor encoders, IMUs (magnetometer, accelerometer, gyroscope), GPS, range sensors (LiDAR, radar, IR, sonar), cameras
- Each domain (industrial, logistics, self-driving car, aerial, aquatic, space, etc.) has its own set of commonly used sensors - for any domains you're looking for roles in, familiarize yourself with the typical combinations used for those applications and how they complement each other
- Considerations / pitfalls - outliers, time synchronization, aligning coordinate frames

Filtering

Things to know:

- What filtering is and why it's necessary
- Simple filters: moving averages, low pass / high pass filters, complementary filters
- Kalman Filters
    - What it is conceptually, when to use one [knowing the derivation / math is a nice plus, but generally won't be expected of you]
    - Common variants (EKF, UKF), when each one is applicable
    - Experience applying one using a library for a real application is a big plus, though not a necessity for non-estimation-specific roles
- Particle Filters [less critical than Kalman Filters but still good to know]
    - What it is conceptually, when you might use one
    - Experience writing one or using a library (nice as a bonus)

Exercises

For this section and the following sections, I strongly recommend (a) setting up some kind of high quality simulation, or (b) building an actual (simple) physical robot with a few different sensors - that you can use to practice the concepts from this guide. Having actual sensor data (real or simulated) to work with, an environment to move around in, and real tasks to try and accomplish will be invaluable for practicing and demonstrating these skills. [TODO - link to portfolio project later in the guide that explains this in more detail]

1. Simulate a noisy IMU (or ideally, record data from a real one) - apply a moving average, low-pass filter, and complementary filter to produce smooth estimates for each IMU sensor. Plot the raw data and the smoothed data and compare
    a. If using a real sensor, analyze some recorded data to understand its noise characteristics. Read through the datasheet and see if there's any guidance on calibration.
2. Simulate a two-wheeled mobile robot
    a. Start by sketching out its geometry and defining the transforms between the center and the wheels. Define the radius/circumference of the wheels and specify the number of encoder ticks in one revolution
    b. Write a function to run one timestep of your simulation. It should accept a 'commanded velocity' (in ticks/second) for each wheel, a timestep duration, and update the pose of the robot accordingly
    c. Add noise to the simulation - give each encoder a noise distribution with some bias and variance
    d. Command your robot to drive in a straight line - run the simulation 100 times and track its trajectory at each timestep for each simulation run. Plot each trajectory and see how the distribution compares to the 'expected' trajectory
3. Apply an EKF to a real-world problem or simulation of your choosing - use a physical robot if you have one, or find a simulator that gives sensor data from multiple sensor streams. Configure an EKF to fuse the different sensor streams into a single smoothed estimate

Interview Questions

- What's the difference between noise, bias, and drift in sensor data?
- Explain the difference between dead reckoning and absolute measurements. What are the strengths and weaknesses of each?
- How does a Kalman Filter work (at a high level)?
- When would you use an EKF vs a UKF?
- What's an application where a particle filter would be useful? Why are PFs often difficult to use in practice?
- What does the covariance matrix represent in a filter?
- What are a few sensors you're familiar with that are useful in [domain of job you're applying to]? How can those sensors be used to complement each other / which ones are most critical?
- Why is time synchronization between sensors critical in sensor fusion?
- What are some examples of relative vs. absolute sensors?
- Suppose your mobile robot only has wheel encoders. What happens to your position estimate over time?
- If you add GPS to that robot, how would the fused estimate behave differently?
- You have an IMU and camera on a drone. What does each sensor contribute to state estimation, and why is combining them useful?
- How would you configure the process noise vs. measurement noise parameters in an EKF? What happens if you set them too low or too high?
- Your GPS signal is noisy, but your odometry is drifting. How would you fuse the two?
- You run an EKF and notice your robot's position estimate lags behind reality. What might be wrong?
- If your fused estimate "jumps" when a sensor drops out, what are some possible causes?
- How would you detect that one of your sensors is producing faulty/outlier data?
- Your IMU and camera have a 10 ms timestamp offset. How might this affect your fused trajectory?
- If your robot's state estimation is unreliable, what downstream effects might this have on the rest of the system?
- If you could only have one sensor on a ground robot, which would you pick, and what tradeoffs would you face?

Localization & Mapping

Use Cases & Motivation:

The topics in this section should help us answer the following questions:

- How do we determine the position of a robot relative to a known map?
- How do we *construct* a map of the environment and determine position at the same time?
- How do we handle updates to maps as the environment changes?
- How do we represent maps so that navigation stack, other systems can work with them?

Key Topics

Odometry & Dead Reckoning

Things to know:

- Core idea - integrating motion over time to determine position changes
- Sensors used for dead reckoning - encoders, IMUs, visual odometry
- Sources of error in this process and how they accumulate over time
- How odometry / DR fit into the overall localization solution, why we still use them despite the shortcomings

Sensor Fusion

Things to know:

- [concepts from the earlier sensor fusion section]
- Common sensor packages used for localization, when to use each, strengths and weaknesses
- Handling common issues - time sync between sensors, calibration, different sensor latencies, temporary sensor failure / occlusion

Map Representations

Things to know:

- Metric maps: occupancy grids, elevation maps, point clouds
- Graph-based maps
- Semantic maps - adding labels / meaning to other map types
- Strengths and weaknesses of each map type, how they can be combined and layered, which application or downstream functions call for which types

SLAM Fundamentals

For non-localization-specific roles, you won't be expected to know more than the basic concepts here, so it's up to you how in-depth you want to go. Developing deep SLAM expertise will look great on a resume and open up many new opportunities, but it's a tricky subject and will require some effort.

Things to know:

- Motivation - why we need SLAM, why it's challenging
- Core concepts: motion models, measurement models, map representation, data association, loop closure (be able to explain these at a high level)
- Common approaches (ability to explain conceptual differences between these approaches, strengths/weaknesses is a nice-to-have, not required for most roles):
    - Filter-based SLAM
    - Graph-based SLAM (recommended starting point)
    - Visual SLAM
    - LiDAR SLAM

Tools / Software

- ROS Navigation Stack, mapping-related message types
- AMCL
- SLAM packages: Cartographer, gmapping, RTAB-Map

Exercises

1. For each of the following scenarios, research and describe the approach you would start with for building a localization system (what sensors, map representations, algorithms, etc. would you use?):
    a. An autonomous surface vessel navigating on the ocean
    b. A mars rover
    c. A pizza-delivery drone
    d. A robotic vacuum for apartments
    e. A robotic lawn mower
2. Find a source of IMU data - either an IMU you buy, your phone, or a dataset you find - and integrate the accelerometer/gyro data to estimate position and orientation over time. Compare the 'dead reckoning' result to the actual position over time.

    Alternatively, try this with a mobile robot with encoder data - drive around, record the encoder ticks and estimate the position over time based on the wheel size and robot geometry. Compare the dead reckoning estimate to the actual position.
3. Either in simulation or with a physical robot, add an absolute measurement source (GPS, fiducial markers) and fuse this with odometry data using a Kalman filter. See if you can get a stable, accurate position estimate over several minutes of navigation.

4. Create a simulation of a robot with a range scanner. Start with a scenario where you have a known map of the environment. Using nothing but the range scans, see if you can localize the robot relative to the map.
5. Experiment with a SLAM package - I recommend Cartographer as a starting point but feel free to experiment with whatever seems interesting. Can you modify your software from #4 so that the robot first builds a map of the environment?

Interview Questions

- What is odometry, and why does it drift over time?
- How do wheel encoders estimate distance traveled? What assumptions do they make?
- How does visual odometry differ from wheel odometry? What are its strengths/weaknesses?
- Can you give an example of when IMU-only dead reckoning would fail quickly?
- Describe a common GPS + IMU fusion setup. What problem does each sensor help solve?
- How would you handle a situation where one sensor fails temporarily (e.g., GPS dropout)?
- What is an occupancy grid?
- Compare metric, topological, and semantic maps. What are some use cases for each?
- When might a 2.5D elevation map be more useful than a 2D occupancy grid?
- What are some downsides of dense point cloud maps compared to occupancy grids?
- How would you handle mapping in a dynamic environment where furniture moves?
- What is the “chicken-and-egg” problem in SLAM?
- Why is loop closure important in SLAM?
- What’s the difference between filter-based SLAM (EKF, FastSLAM) and graph-based SLAM?
- Give an example of a robot that can get by with pure localization, and one that requires SLAM.
- What challenges arise when running SLAM in real-time on a resource-constrained robot?
- How would you debug a robot that consistently thinks it’s in the wrong part of the map?
- How would you check whether your localization system is properly time-synchronized?
- How might localization requirements differ between a warehouse robot, a Mars rover, and a home vacuum robot?
- If you were asked to reduce localization drift without adding new sensors, what strategies might you try?
- What tradeoffs do you consider when choosing between 2D vs 3D mapping approaches?
- In a multi-robot system, how could you merge maps from different robots?

Perception

This entire section falls under the category of 'nice-to-have'. This is a huge topic and has many specializations within it. I recommend becoming familiar with the core problems we're trying to solve with perception and some of the common approaches we use to solve them. Focusing in particular on how perception fits within the rest of the system, and how issues with perception might affect other subsystems. And as always, gaining deeper knowledge here is a great way to stand out.

Use Cases & Motivation

The topics in this section should help us answer the following questions:

- What objects are around the robot? Where are they? How are they moving?
- What is the shape of the environment/terrain - where is the ground? What parts of the environment are traversable?
- How can I interact with the objects around me? What can I touch, grasp, push? Are there any hazards?

Key Topics

Sensor foundations

Things to know:

- Types of sensors most commonly used for perception tasks, strengths/weaknesses of each
    - Cameras - monocular, stereo, RGB-D, fisheye & other lens shapes
    - LiDAR - 2D, 3D, solid state vs. spinning
    - Radar, sonar - what environmental challenges they help with
- What form of data output each of these produces, basic tools for viewing and working with it

Vision / Geometry basics

Things to know:

- Image processing basics: edge detection, feature detection, optical flow
- Point cloud processing basics - filtering, sampling, normals, registration

Objects

Things to know:

- Detection/classification - identifying discrete objects within images or point clouds, creating 2D or 3D bounding boxes

- Segmentation - organizing pixels / points by what class(es) of object they belong to
- Tracking & prediction - correlating objects across multiple frames, anticipating trajectories, determining intent

Environment-state

Things to know:

- Differentiating navigable / non-navigable terrain
- Representing the environment - occupancy grids, semantic maps

Exercises

1. Take some pictures with your phone and experiment with several image processing operations (smoothing, edge detection, thresholding, feature extraction)
2. Find datasets of LiDAR, Radar, and depth camera data. Practice loading the data programmatically, visualizing it, and manipulating it in some way
3. Record a video from the perspective of a robot driving along the ground in your apartment. See if you can
    a. Identify the ground plane in the video frames
    b. Determine the distance to different points within the frames
    c. Identify objects sitting above the ground plane
    d. Construct a local occupancy grid for the region in front of the robot, separating driveable areas from non-driveable areas
4. Research image segmentation approaches and apply one to a video you record. See if you can identify different object types or extract other useful information about the scene
5. Record a video with a person walking around, a tennis ball bouncing, or some other identifiable object moving around the scene. Use a Kalman filter or other object tracking technique to track the motion of the object throughout the scene

Interview Questions

- What are the tradeoffs between using a monocular camera, stereo camera, RGB-D camera, and LiDAR for perception?
- How does LiDAR generate a 3D point cloud? What are some common limitations?
- In what scenarios might radar outperform LiDAR or vision systems?
- What is the difference between detection, segmentation, and tracking? Give an example of when each is used.
- How would you go about removing the ground plane from a LiDAR point cloud?
- Can you explain the difference between semantic, instance, and panoptic segmentation?
- How do you associate object detections across frames to build a consistent track?
- What role does a Kalman filter (or similar) play in multi-object tracking?
- How do you handle occlusion when tracking multiple objects in a scene?

- Explain a general approach for detecting drivable free space in a camera image
- Suppose your robot's perception system works well in the lab but fails outdoors in bright sunlight. What could be going wrong?
- How would you benchmark or evaluate the performance of a perception system? What metrics would you use?
- How would you structure a perception pipeline that supports both navigation and manipulation tasks?

Planning & Navigation

Use Cases & Motivation

The topics in this section should help us answer the following questions:

- What path should the robot take to get from point A to point B?
- How does the robot avoid objects in its vicinity
- Once we know the path the robot should follow, how do we drive the motors in order to follow that path?
-

Key Topics

General Concepts

Things to know:

- Difference between *planning, control,* and *navigation* and how they connect
- Difference between *global planning* and *local planning*
    - Assumptions about static vs dynamic environment
    - Re-planning frequency for each
    - Optimality vs. reactivity tradeoff
    - Typical data representations for maps, obstacles, routes, and trajectories
- How each of these concepts applies (or doesn't) to a few different robot contexts, e.g. mobile robot, robot arm, satellite
- Be able to draw a simple box diagram showing how all the parts of the navigation stack interact

Map Representations:

Things to know:

- What each of these map representations is, how to implement one, and when it might be used
    - Occupancy grids, elevation maps, costmaps
    - Graph based maps
    - Semantic maps
- How we expand maps to 2.5D or 3D and what additional challenges this poses
- Difference between static and dynamic maps, when to use each, and how we can combine them
- Tradeoffs - map resolution vs memory vs speed
- Be able to think through a variety of robot contexts and choose reasonable map types for those contexts

Planning Algorithms

Things to know

There are many, many planning algorithms - I've tried to cover some of the most common ones here. My recommendation is that you be able to explain how most of these work conceptually, choose one that's appropriate for a given application, and have experience implementing one or a few algorithms from each grouping. But you don't need to be an expert on everything here or implement all of these.

- Graph search algorithms
    - Non-heuristic - depth first search, breadth first search, Dijkstra's
    - Heuristic: A*, D* / D* Lite, Theta*
    - Understand the best and worst cases for each, when you'd choose one vs. the others
    - Be able to apply to a grid-based map or a graph-based map
- Sampling-based planners
    - RRT, RRT*, PRM
    - Understand what problems sampling-based planners solve (high dimensional, continuous planning spaces), be able to give examples of where they're useful
- Trajectory planning
    - Distinction between path / trajectory, when we need to plan trajectories
    - Trajectory representations: waypoints with timestamps, splines
    - DWA, TEB, Trajectory Rollout, MPC

Constraints & Safety

Things to know:

- How to factor robot footprint into planning
- Safety margins - how to include them, what the tradeoffs are
- Nonholonomic constraints - what they are and how to handle them in planning algorithms
- Dynamic limits - max velocity/acceleration/jerk, how to ensure the robot can physically follow the plan

Exercises

1. Implement a global planner for an occupancy grid
    a. Create a 'map generator' that creates occupancy grids with obstacles in different locations
    b. Plan routes between arbitrary sets of points. Experiment with a few different algorithms, compare planning speeds
    c. Implement some kind of smoothing on your paths so that they avoid sharp corners when possible
    d. Add a step that converts your occupancy grid to a costmap - see if you can achieve routes that don't 'hug' the obstacles and instead keep a safe distance
2. Implement a sampling-based planner for a continuous environment
    a. Generate a set of randomly placed, circular obstacles with different radii

    b. Write a function that checks if a point in the world is in collision with one of the obstacles
    c. Implement an RRT or PRM planner to compute paths between different points in the environment
3. Implement a trajectory generator for a simulated two-wheeled robot
    a. Start with your solution from exercise #1. Place your robot at a random point along the route. Compute a local target position at some fixed distance forward along the route.
    b. Use DWA or trajectory rollout to compute possible trajectories and select the best one
    c. Test with different starting conditions (starting velocity, position/orientation relative to the path), and with different constraints on the robot's movement (e.g. max turn radius)
4. Build a motion planner for a simulated robot arm - use your simulated 2 or 3 DOF arm from earlier.
    a. Expand the simulation to include randomly placed circular obstacles. Assume we only care about collisions between the end-effector and the obstacles
    b. Plan end-effector paths between different points using an algorithm that seems appropriate for this application
    c. Can you guarantee that any computed paths are physically reachable by the arm?
    d. Can you factor in different 'preferences'? For example, say you prefer to keep the arm less-extended whenever possible, or that you want to stay far clear of any obstacles unless there's no other option. How can you incorporate these preferences into your planner?

Interview Questions

- Explain the roles of a global planner vs. a local planner. Why do we usually need both?
- What are the main tradeoffs between optimality and reactivity in planning systems?
- How does an occupancy grid work, and what are its limitations?
- What is a costmap? Why do we "inflate" obstacles in costmap?
- Compare grid-based maps vs. topological maps. When would you use each?
- Why might a 2D map be insufficient for certain robots, and what alternatives exist?
- Walk me through how A* works. What makes it different from Dijkstra's algorithm?
- What problem does D* solve that A* does not?
- What's a situation where you'd use a sampling-based planner? What's an example of a sampling-based algorithm you're familiar with and how does it work?
- How does the Dynamic Window Approach (DWA) work? What are its limitations?
- What's the role of forward simulation in local planners?
- How do local planners account for robot kinematics and dynamics?
- How would you model a robot's footprint for planning? Why does the choice of model matter?
- What's the difference between a kinematic constraint and a dynamic constraint? Give an example of each.
- How do planners handle narrow passages in the environment?
- What is the role of safety margins in navigation? How do you choose them?
- How does the global planner interact with the local planner and controller in a typical navigation stack?
- How often does a global planner need to replan compared to a local planner? Why?

Control

Use Cases & Motivation

The topics in this section should help us answer the following questions:

- Once we have a trajectory plan, how do we choose velocity commands to actually follow that trajectory?
- Once we have a velocity command, how do we apply current to our motor so it actually holds the correct rotation speed?
- How can we get a quadcopter or a robot arm to hold a specific position?
- How do we prevent oscillation, overshoot, or instability in these control problems?

Key Topics

Path Following / Execution

Things to know:

- Be able to explain the basic problem framing - we have a local path or trajectory to follow and know the current robot position and speed, how do we turn this into a command our motor controllers can use?
- Terminology - cross-track (lateral) error, heading error, along-track (longitudinal) error
- Geometric path following controllers - pure pursuit, Stanley controller, vector field methods
- Implementation considerations - smoothing / curvature constraints, lookahead distance tuning, stability vs. responsiveness
- How to apply these concepts to different robot types - ground vehicle, drone, robot arm

Control Theory (Basics)

Things to know:

- Be able to draw a simple box diagram for a control system with setpoint, noise, measurement, command, etc. and be comfortable talking through each part conceptually
- Be able to talk through these concepts / terminology:
    - Stability (stable, unstable, marginally stable)
    - Transient response vs steady-state response
    - Rise time, overshoot, settling time, steady-state error
    - Feedback vs. feedforward control
    - Disturbances, noise
    - Linear vs. nonlinear systems
- Understand what characteristics a ‘good’ controller has vs. a bad one and talk through what things you would look for when analyzing one to understand its peformance

For an entry-level / non-control-specific role, having a conceptual grasp on these is sufficient, no need to be an expert in the math. Focus more on implementation and practical details, only go deeper in the theory if you want to specialize in this area or find it especially interesting.

PID Control and beyond

Things to know:

- What a PID controller is, what some applications are, why they're so useful
- Be able to explain what each parameter means, what role it plays in the controller
- Know some of the common variants (PI, PD, PI-D / I-PD) and when it makes sense to use each one
- PID tuning - manual tuning, Ziegler Nichols, automatic tuning approaches
- Tuning tradeoffs: response time vs stability vs steady-state accuracy
- Beyond PID:
    - Adding feedforward terms
    - LQR, MPC, Adaptive control

Practical Considerations & Pitfalls

Things to know:

- Actuator saturation, integrator windup - know what these are, how to spot them, how to avoid them
- Update rates - rules of thumb for choosing a loop rate, how update rates vary for different applications
- Sensor noise - understand how noise can affect control systems, trade-off between signal smoothing and responsiveness
- Understand where latency is introduced to control systems and the problems this can cause
- Stability vs performance trade-off, how to choose the right balance
- Safety - how to build in fail-safes and handle disturbances gracefully

Exercises

1. Simulate a pair of wheel controllers for a two-wheeled robot
    a. Start by simulating a single motor + wheel. Write a function that takes a current or voltage setting and outputs a rotation speed (using any mapping that seems reasonable here). Write another function that takes the rotation speed, wheel radius, and time elapsed, and outputs the distance travelled by the wheel
    b. Next, simulate the second wheel. Then, update your simulation to track the position of the center of the robot over time, given the individual wheel speeds.

    c. Modify your simulation so that the friction / driving difficulty varies by where the robot is in the world. Add a ‘driving difficulty’ layer to your map, and when mapping from motor voltage → rotation speed, factor in the difficulty so that more difficult terrain requires higher voltage to maintain the same rotation speed
    d. Now your ready to write your controller - add a controller to each motor such that you can set the desired rotation speed (or wheel velocity) for that motor and it will maintain the right speed, regardless of the terrain
    e. Next, write a function that lets you specify an overall linear and angular velocity for the center of the robot. The function should convert the linear/angular velocity to speed commands for the individual wheels and set their controllers accordingly.
    f. At this point, you should be able to specify a linear and angular velocity, let the simulation run, and the robot should follow a smooth arc, regardless of the terrain
2. Expand your simulation from #1 so that the robot can follow a path
    a. Implement a local planner that generates an instantaneous trajectory for the robot to follow, given its current position and velocities and a target position
    b. Implement a path-following algorithm that takes the trajectory and the current position of the robot, and outputs an instantaneous linear and angular velocity for following the path
    c. Connect all the pieces and debug until you’ve got a robot that can drive to different target locations within your simulated environment!
3. Build a real-world control system for an application of your choice. You’ll need some kind of sensor, some kind of actuator, and something to run your control software on. Log the measured value, target value, and error over time. Experiment with different controllers, parameters, and see if you can build a stable controller. Create integrator windup or actuator saturation and then practice mitigating them.
    a. Temperature controller
    b. Inverted pendulum
    c. Water-level controller
    d. Fan motor speed controller

Interview Questions

- In lane keeping for autonomous cars, why do we care about both cross-track error and heading error?
- Can you explain the difference between open-loop and closed-loop control, and give a real-world example of each?
- What does it mean for a control system to be stable?
- Define rise time, overshoot, settling time, and steady-state error. Why do they matter?
- What’s the difference between feedforward and feedback control, and when might you use each?
- Describe the effect of the P, I, and D terms in a PID controller.
- Why might you prefer PI control over full PID in many industrial systems?
- What problem does the integral term solve, and what new problem does it introduce?
- How does derivative action help in control, and why is it often noisy in practice?
- Suppose your system overshoots badly — which PID term would you adjust first, and why?
- How would you tune a PID controller if you had no model of the system?

- Why does actuator saturation cause issues in control systems?
- How does control loop update rate affect system stability and performance?
- If your system is stable in simulation but unstable on real hardware, what real-world effects might be missing from your model?
- What kinds of nonlinearities do real actuators often introduce into control systems?
- Walk me through how you'd design a controller for a mobile robot that needs to follow a curved path at constant speed.
- Imagine your robot arm oscillates when trying to reach a target position. What diagnostic steps would you take to identify the problem?
- You're asked to stabilize a drone in hover, but you notice slow altitude drift. Which control terms or strategies would you adjust?
- How would you explain the importance of cascaded control loops to a non-technical teammate?

Simulation & Testing

You should understand the basic ideas around why we simulate and what a typical simulation progression looks like, but most of the details in this section are ‘nice-to-haves’. The specific simulation tools used will vary significantly by domain so I’ve mostly stuck to general purpose tools and high-level concepts here.

With that being said, if you’re working to refresh or expand your robotics knowledge, developing your simulation skills is extremely useful. If you can create (even relatively simple) simulations, you can practice almost all of the other subjects covered in this guide without spending a dime on hardware. So in my opinion, building up a small set of simulations where you can test out different algorithms is a great place to start if you expect to spend a few months prepping for job interviews.

Use Cases & Motivation

The topics in this section should help us answer the following questions:

- How can we systematically test a robot when physical experiments are slow, expensive, or dangerous?
- How can we validate a robot’s control or perception pipelines across a large number of scenarios, rather than select demos?
- What aspects of a robot’s behavior can a simulation capture well? What aspects are difficult to capture with simulation

Key Topics

Simulation fundamentals

Things to know:

- Why simulation is useful - speed of iteration, safety, scale, determinism/reproducibility
- Limits of simulation:
    - Many physics aspects are simplified or ignored, both on/in the robot and in the environment
    - Simulated sensors don’t perfectly replicate real-world sensor data
    - Environment is simplified in sim, missing things you’ll encounter in the real world
    - Hardware interfaces, network setup may be different or ignored in sim
- Typical sim / test progression - understand what types of issues each of these layers catches, why this ordering is useful
    - Unit tests: software only, test narrow software components, run on every commit
    - SIL (software-in-the-loop): full stack simulation in a virtual world, no real hardware
    - HIL (hardware-in-the-loop): full stack simulation, some real hardware used
    - Field tests: Use physical robot in controlled conditions, escalate from simple to complex

Tools / Software

I've tried to include a variety of commonly used tools here, but many domains (e.g. space, drones, self-driving cars, manipulators) will have their own set of domain-specific simulation tools. I recommend starting with Gazebo or one of the other general-purpose simulators, and if you're targeting a specific domain, research the tools that are most often used for that domain and focus on those next.

Simulators

- Gazebo - great general purpose starting point, easy integration with ROS
- Other general purpose sims: Webots, CoppeliaSim, Mujoco
- Specialized / high end: NVIDIA Isaac Sim, Unreal Engine, CARLA, AirSim

Other tools

- Robot description / modelling: URDF, Xacro, SDF
- Visualization: RViz, Foxglove Studio, PlotJuggler
- Data capture & replay: rosbag2

Modeling robots

Things to know:

- Representing the robot - Meshes (STL), URDF
- What aspects of the robot should be modeled - links, joints / joint types, mass, actuators, sensors
- Modeling actuators - torque/velocity limits, effort curves
- Modeling sensors - know the parameters for a variety of sensor types (e.g. for LiDAR: # of beams, scan rate, range, angular resolution), how to model noise
- How to measure specs, actuator/sensor characteristics from real hardware, use it to improve simulation fidelity
- Be able to talk through some of the things robot/sensor/actuator models miss and what problems this can cause

Testing terminology

Be able to define each of these and give examples

- Unit testing - tests individual software components
- Integration testing - tests multiple components working together
- System (end-to-end) testing - running the full software stack on robot (or sim) against typical tasks
- Scenario testing - structured environments/tasks that target specific failure modes
- Regression testing - a fixed set of tests/scenarios that each new commit is run against to ensure that changes don't re-introduce old bugs

- Acceptance testing - testing the system against predefined requirements or specs

Exercises

Rather than go through a bunch of simulation-specific exercises, my suggestion here is to, first, work through a few simple examples/tutorials with Gazebo (or a different sim tool if there's a specific one you want to focus on) just to learn the basics of that tool. Then, as you work through other sections of this guide, practice testing the algorithms you're working on in that sim tool.

You can start simple (very basic robot, no sensor noise, flat environment) and gradually increase the complexity as you want to test different things. For example if you're working on filtering, you can start modelling the noise in each of your sensors - or if you're testing out planning algorithms, you can start spawning different obstacles or constructing scenarios to test difficult cases.

Interview Questions

- What are some reasons to simulate before testing on real hardware? What kinds of problems can simulation *not* catch?
- What are the advantages of Gazebo/Ignition compared to tools like Mujoco or Isaac Sim?
- If your goal was to generate photorealistic datasets for training a perception model, which simulator(s) would you use and why?
- What properties should you include in a robot's URDF/SDF for realistic dynamics?
- Why is it important to separate visual meshes from collision meshes?
- Suppose you're simulating a LiDAR. What parameters and noise models would you include to make it realistic?
- How would you simulate IMU bias drift? Why does it matter for testing algorithms?
- What's the difference between unit, integration, and regression testing?
- Give an example of a bug you'd expect to catch in simulation, versus one you'd expect to only catch in the field.
- What's the value of log replay, and how would you integrate it into your testing workflow?
- Define Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL). What kinds of issues are surfaced by HIL that SIL cannot?
- Walk me through a typical progression from simulation to field deployment.
- How would you prevent overfitting to simulation in your robot software?
- Explain some uses of randomization in simulation
- If your robot passes every test in simulation but fails in the field, how would you systematically debug the issue?

Middleware & Tooling

Use Cases & Motivation

The topics in this section should help us answer the following questions:

-

Key Topics

Topic 1

Things to know:

-

Exercises

6.

Interview Questions

- Pub/sub model
- Message passing, serialization
- ROS/ROS2, MQTT
- Tools for plotting / logging sensor data
- Tools for visualizing robot state / environment (Rviz, other)
- Visualizing transform trees
- Quality of Service core concepts

Interfaces

Use Cases & Motivation

The topics in this section should help us answer the following questions:

-

Key Topics

Topic 1

Things to know:

-

Exercises

7.

Interview Questions

- Serial communication: CAN, UART, SPI, I²C, USB, Ethernet
- Sensor calibration
- Device driver fundamentals & debugging
- Data rates, accuracy vs rate considerations

Reliability & Safety

Use Cases & Motivation

The topics in this section should help us answer the following questions:

-

Key Topics

Topic 1

Things to know:

-

Exercises

8.

Interview Questions

- Watchdogs, timeouts, monitoring
- E-STOPs
- Health/status monitoring
- Best practices
    - Simulate first whenever possible
    - ‘Real world’ edge cases
    - Safety critical vs not
    - Safe default states, simple human overrides
    - Constrained environments

Emerging / State of the Art

- Neural networks based perception, object detection
- Imitation learning / reinforcement learning
- Vision-language-action models

Specializations:

Software domains:

- Each of the categories above can be its own specialization
- Machine learning for robotics
- Manipulation / grasping
- Multi-robot coordination
- Human-robot interaction

Application Domains:

- Aerial robotics
- Aquatic robotics
- Ground-based mobile robots
- Space systems
- Industrial automation / manufacturing
- Logistics / warehouses
- Agriculture
- Hazardous environment / exploration
- Consumer robotics
- Defense
- Medical / surgical
- Construction / infrastructure
- Domestic robots