

Practical Robotics Kinematics Exercise Manual

A Real World Engineer's Guide

Practical Robotics Kinematics Exercise Manual

A Real World Engineer's Guide

First Edition

Gareth Morgan Thomas
Auckland, New Zealand



Published by Burst Books
Auckland, New Zealand

Copyright © 2025 Gareth Morgan Thomas
All rights reserved.

About the Author

Gareth Morgan Thomas is a qualified expert with extensive expertise across multiple STEM fields. Holding six university diplomas in electronics, software development, web development, and project management, along with qualifications in computer networking, CAD, diesel engineering, well drilling, and welding, he has built a robust foundation of technical knowledge. Educated in Auckland, New Zealand, Gareth Morgan Thomas also spent three years serving in the New Zealand Army, where he honed his discipline and problem-solving skills. With years of technical training, Gareth Morgan Thomas is now dedicated to sharing his deep understanding of science, technology, engineering, and mathematics through a series of specialized books aimed at both beginners and advanced learners.

Chapter 1

Why Engineers Don't Code Kinematics From Scratch

1.1 The Reality Check

1.1.1 When your manager says "just make the robot move"

[When your manager says "just make the robot move"]

[Motor Control Basics] Exercise: A differential-drive robot has two wheels controlled by separate DC motors.

1. Write a Python function `set_motor_speed(left_pwm, right_pwm)` that sends PWM signals to the motors using the GPIO library. Assume a 12-bit PWM resolution.
2. The left motor spins backward when given positive PWM values. Modify your function to invert the left motor's polarity.
3. Add dead-zone handling to ignore PWM values below 5% of the maximum duty cycle.

[Odometry from Encoders] Exercise: A robot has wheel encoders with 20 ticks per revolution and wheel diameter of 0.1 m.

1. Derive the formula to convert encoder ticks to linear distance traveled.
2. Implement the formula in Python as `get_distance(left_ticks, right_ticks)`.
3. Extend the function to return both linear displacement and heading change (in radians) given a wheelbase of 0.3 m.

[Obstacle Avoidance] Exercise: A robot has three front-facing IR proximity sensors (left, center, right) returning values 0–1023.

1. Write pseudocode to stop the robot if `center_sensor` exceeds 800.
2. Implement a wall-following behavior that steers away when `left_sensor` exceeds 600.
3. Add a 5-second timeout that overrides all behaviors if no sensor update occurs.

1.1.2 Time-to-market vs reinventing the wheel

[Time-to-market vs reinventing the wheel]

Software Development Trade-offs Exercise: A startup is building a mobile app for food delivery and must decide whether to:

1. Implement a custom geolocation tracking system from scratch
2. Integrate Google Maps API with a monthly cost of \$500
3. Use an open-source alternative with limited documentation

Calculate the break-even point in months if:

- Custom development would take 3 months with \$15,000 labor cost
- The open-source option requires 2 weeks integration but may need \$2,000 in contractor support
- The team has \$20,000 total budget

Component Reuse Analysis Exercise: Given this Python code for a custom CSV parser:

```
def parse_csv(file_path):  
    data = []  
    with open(file_path) as f:  
        for line in f:  
            data.append(line.strip().split(','))  
    return data
```

1. Identify three edge cases this parser would fail to handle
2. Compare its performance against Python's `csv` module for 10MB files
3. Modify the function to properly handle quoted fields containing commas

Third-Party Library Evaluation Exercise: You need to add user authentication to a web app. Research two options:

- `django-allauth` vs `python-social-auth`
- List three critical security features to verify in either library
- Create a feature matrix comparing setup time, maintenance overhead, and OAuth provider support
- Estimate the time savings vs building a custom solution with `bcrypt` and session management

1.1.3 The hidden complexity of "simple" kinematics

[The hidden complexity of "simple" kinematics]

[Bicycle wheel angular velocity] Exercise: A bicycle wheel with radius $r = 0.35$ m rolls without slipping on a flat surface. The wheel's center moves with constant velocity $v = 5$ m/s.

1. Calculate the angular velocity ω of the wheel in radians per second.
2. Determine the linear velocity of a point at the top of the wheel relative to the ground.
3. Find the angle (in degrees) through which the wheel turns in 2 seconds.

[Robot arm inverse kinematics] Exercise: A 2D robotic arm has two links with lengths $L_1 = 0.5$ m and $L_2 = 0.3$ m. The end effector must reach position $(x, y) = (0.6$ m, 0.4 m).

1. Write the forward kinematics equations for this arm.
2. Derive the inverse kinematics solution for joint angles θ_1 and θ_2 .
3. Calculate the numerical values of both possible solutions for the joint angles.

[Pendulum energy conservation] Exercise: A simple pendulum of length $l = 1.2$ m has a bob of mass $m = 0.4$ kg. It is released from rest at an initial angle $\theta_0 = 30$ degrees.

1. Calculate the maximum potential energy of the system.
2. Determine the maximum speed of the pendulum bob.
3. Write a Python function using `scipy.integrate.solve_ivp` to simulate the motion.

```
def pendulum_simulation(theta0, t_span):
    # Your implementation here
```

1.2 Library Landscape and Trade-offs

1.2.1 Performance benchmarks: Speed vs ease of use

[Performance benchmarks: Speed vs ease of use]

Matrix Multiplication Optimization Exercise:

1. Implement a naive matrix multiplication function in Python using nested loops for two 1000x1000 matrices. Time its execution.
2. Rewrite the function using NumPy's `dot()` method and compare the execution time.
3. Modify the NumPy implementation to use `einsum()` instead and measure any performance difference.
4. Explain why the NumPy versions outperform the naive implementation in terms of memory access patterns.

Database Query Optimization Exercise:

1. Create a SQLite database with a table `customer_orders` containing 1 million rows of mock order data.
2. Write a query to find the top 10 customers by total purchase amount without any indexes. Record the execution time.
3. Add appropriate indexes to the table and rerun the query. Compare the execution times.
4. Write an equivalent query using a CTE (Common Table Expression) and compare its performance to the indexed version.

JSON Parsing Performance Exercise:

1. Generate a 50MB JSON file with nested arrays of employee records using Python's `json` module.
2. Write a script to parse this file using Python's standard `json.load()` and measure the memory usage.
3. Rewrite the parser using `ijson` for streaming parsing and compare the memory consumption.
4. Implement the same parsing task in Rust using `serde_json` and compare the execution speed to both Python implementations.

1.2.2 ROS ecosystem (MoveIt, KDL) - when you're already in ROS

[ROS ecosystem (MoveIt, KDL) - when you're already in ROS]

Forward Kinematics with KDL Exercise: Implement a forward kinematics solver for a 3-DOF robotic arm using KDL in ROS.

1. Create a URDF file for a robotic arm with three revolute joints (`joint1`, `joint2`, `joint3`) and links of lengths 0.5m, 0.3m, and 0.2m.
2. Write a ROS node that initializes a KDL chain from the URDF using the `kdl_parser`.
3. Compute the end-effector position for joint angles (0.5, 0.2, 0.1) radians using `KDL::ChainFkSolverPos_recursive`.
4. Print the resulting end-effector position in the terminal using `ROS_INFO`.

MoveIt! Motion Planning Exercise: Configure MoveIt! to plan trajectories for a UR5 robot in a cluttered environment.

1. Install the `ur_description` package and generate a MoveIt! configuration for the UR5 robot.
2. Add three box obstacles to the planning scene using `moveit_msgs::CollisionObject`.
3. Create a ROS node that moves the end-effector from position (0.3, 0.2, 0.5) to (0.3, -0.2, 0.5) in Cartesian space.
4. Verify the planned path avoids collisions using `MoveItVisualTools`.

ROS-Control Integration Exercise: Implement a simple joint position controller using ROS-Control.

1. Create a ROS package with dependencies on `ros_control` and `gazebo_ros_control`.
2. Write a custom hardware interface that reads/writes joint positions from/to `std_msgs::Float64` topics.
3. Configure a `position_controllers::JointPositionController` in a YAML file for `joint1`.
4. Launch the controller and test it by publishing target positions to the `/joint1/command` topic.

1.2.3 Standalone libraries (Drake, Pinocchio) - when you need control

[Standalone libraries (Drake, Pinocchio) - when you need control]

[Forward Kinematics with Pinocchio] Exercise: Using the Pinocchio library, compute the forward kinematics for a 3-DOF robotic arm.

1. Define a simple 3-DOF robot model with revolute joints using `pinocchio.Model`.
2. Set joint angles to `[0.5, -0.3, 1.2]` radians.
3. Compute the end-effector position using `pinocchio.forwardKinematics`.
4. Print the resulting end-effector position in world coordinates.

[Trajectory Optimization with Drake] Exercise: Solve a trajectory optimization problem for a double integrator system using Drake.

1. Create a `Drake.MultibodyPlant` with a single floating mass.
2. Define a quadratic cost for state deviation and control effort.
3. Set initial state to `[1.0, 0.0]` and target state to `[0.0, 0.0]`.
4. Solve the optimization over a 5-second horizon using `Drake.Solve`.
5. Plot the resulting state and control trajectories.

[Collision Detection with Pinocchio] Exercise: Implement collision checking between two simple shapes using Pinocchio.

1. Create two sphere geometries with radii 0.5 and 0.3.
2. Position them at `[0,0,0]` and `[0.7,0,0]` respectively.
3. Use `pinocchio.computeCollision` to check for collisions.
4. Repeat the check after moving the second sphere to `[0.4,0,0]`.
5. Print whether a collision was detected in each case.

1.2.4 Simulation-first tools (PyBullet, MuJoCo) - prototype before hardware

[Simulation-first tools (PyBullet, MuJoCo) - prototype before hardware]

[PyBullet Simulation Setup] Exercise: Set up a basic PyBullet simulation environment with the following requirements:

1. Initialize the PyBullet physics client in GUI mode using `p.connect(p.GUI)`.
2. Load a URDF model of a simple table from `"table/table.urdf"`.
3. Add a cube primitive of size 0.1m at position (0, 0, 1) using `p.createCollisionShape` and `p.createMultiBody`.
4. Set gravity to -9.81 m/s^2 in the z-axis.
5. Run the simulation for 1000 steps with a fixed time step of 1/240 seconds.

[MuJoCo XML Modeling] Exercise: Create a MuJoCo XML model for a 2-DOF robotic arm with the following specifications:

1. Define two revolute joints with limits of -90 to 90 degrees.
2. Use cylindrical geometries for both links with lengths 0.5m and radii 0.05m.
3. Set joint damping values to 0.1 Nms/rad for both joints.
4. Include a spherical end-effector with radius 0.1m.
5. Add a light source positioned 2m above the base.
6. Validate your model using the MuJoCo visualizer.

[Contact Force Analysis] Exercise: Simulate and analyze contact forces between two objects in PyBullet:

1. Create a ground plane and a box of size $0.2\text{m} \times 0.2\text{m} \times 0.2\text{m}$ at height 1m.
2. Enable contact point calculation using `p.setPhysicsEngineParameter`.
3. Simulate the box falling onto the ground plane.
4. Extract the maximum normal force during impact using `p.getContactPoints`.
5. Plot the force magnitude over time using matplotlib.
6. Repeat with a restitution coefficient of 0.7 and compare results.

1.3 Choosing Your Stack

1.3.1 Decision matrix: Project constraints vs library capabilities

[Decision matrix: Project constraints vs library capabilities]

Constraint Analysis Exercise: Exercise:

1. Create a decision matrix comparing three Python plotting libraries: Matplotlib, Plotly, and Seaborn
2. Use these constraints: mobile compatibility, 3D plotting, and animation support
3. Score each library (1-5) for each constraint, with 5 being best support
4. Calculate weighted scores using weights: mobile (0.4), 3D (0.3), animation (0.3)
5. Recommend the best library for a project requiring all three features

API Limitation Exercise: Exercise:

1. Identify three key limitations of NumPy's `fft` module for real-time audio processing
2. Propose workarounds for each limitation without switching libraries
3. Write a Python function using `numpy.fft` that handles non-power-of-two input sizes
4. Benchmark your function against the standard `numpy.fft.fft` with 1000 random inputs
5. List any remaining constraints after your optimizations

Dependency Conflict Exercise: Exercise:

1. You need both `tensorflow==2.8.0` and `opencv-contrib-python==4.5.5.64` in a project
2. Create a dependency graph showing all conflicting sub-dependencies
3. Propose version changes that maintain maximum functionality for both main packages
4. Write a `requirements.txt` file with your resolved versions
5. Test your solution in a fresh virtual environment and report any remaining issues

1.3.2 Real-world case studies: What companies actually use

[Real-world case studies: What companies actually use]

Database Optimization at Scale Exercise: A social media company stores user posts in a `posts` table with columns `post_id`, `user_id`, `content`, and `timestamp`. The table has 500 million rows. Complete these tasks:

1. Write a SQL query to find the top 10 most active users by post count in the last 30 days.
2. Explain how you would index this table to optimize the query.
3. Design a sharding strategy to distribute this table across 8 database servers.

Log Processing Pipeline Exercise: An e-commerce company needs to process 10GB/hour of server logs stored in `logs/YYYY-MM-DD/HH.json.gz` format. Complete these tasks:

1. Write a Python script using `gzip` and `json` modules to count HTTP 500 errors in a single file.
2. Design a Spark job to aggregate daily error counts by hour and status code.
3. Calculate the storage requirements for 90 days of logs at current volume with 3x replication.

Machine Learning Deployment Exercise: A ride-sharing company wants to deploy a fare prediction model. Complete these tasks:

1. Write a Flask API endpoint that accepts `pickup_lat`, `pickup_lon`, `dropoff_lat`, `dropoff_lon`, and returns a predicted fare.
2. Create a Dockerfile to containerize the model with all dependencies.
3. Design a canary deployment strategy to roll out the model to 5% of traffic initially.

1.3.3 Migration paths: Starting simple, scaling up

[Migration paths: Starting simple, scaling up]

Database Schema Migration Exercise:

1. Create a PostgreSQL table named `customer_orders` with columns: `order_id` (integer, primary key), `customer_name` (varchar(50)), `order_date` (date), and `total_amount` (numeric(10,2)).
2. Write a migration script to add a new column `shipping_status` (varchar(20)) with default value 'Pending'.
3. Update all records older than January 1, 2023 to have `shipping_status = 'Completed'`.

API Versioning Exercise:

1. Implement a Flask route `/api/v1/products` that returns a JSON array of product names.
2. Create a new version `/api/v2/products` that includes additional fields: `product_id` and `price`.
3. Add a deprecation warning header to the `v1` endpoint while keeping it functional.

Load Testing Exercise:

1. Write a Locust script to simulate 100 users accessing `/api/products` endpoint with 10 requests per second.
2. Modify the script to include a 2-second think time between requests.
3. Add assertions to verify response times never exceed 500ms during the test.

Chapter 2

Getting Your Environment Not to Hate You

2.1 The "It Works on My Machine" Problem

2.1.1 Docker containers for robotics development

[Docker containers for robotics development]

[ROS Noetic Container Setup] Exercise:

1. Create a Dockerfile that uses the official `ros:noetic-ros-core` image as a base.
2. Add commands to install the `ros-noetic-desktop-full` package and any required dependencies.
3. Expose port 11311 for ROS master communication using the `EXPOSE` instruction.
4. Build the image with tag `ros-noetic-custom:v1` and verify it runs with `docker run -it ros-noetic-custom:v1`.
5. Launch the ROS master inside the container and verify it responds to `rostopic list`.

[Gazebo Simulation Environment] Exercise:

1. Start with your `ros-noetic-custom:v1` image from the previous exercise.
2. Create a new container with GPU passthrough using `--gpus all` flag.
3. Run `roscore` in one terminal and `roslaunch gazebo_ros gazebo` in another.
4. Spawn a TurtleBot3 model using the `roslaunch turtlebot3_gazebo turtlebot3_world.launch` command.
5. Capture the container state as a new image using `docker commit` for future use.

[Cross-Container ROS Networking] Exercise:

1. Create a custom Docker network named `ros-network` with `docker network create`.
2. Launch two containers: one running `roscore` (named `ros-master`) and another running `rqt_graph`.

3. Configure both containers to use the `ROS_MASTER_URI` environment variable pointing to `ros-master`.
4. Verify the `rqt_graph` container can visualize the ROS master's node graph.
5. Add a third container running a simple publisher node and confirm it appears in `rqt_graph`.

2.1.2 Version hell and how to avoid it

[Version hell and how to avoid it]

Dependency Conflict Resolution Exercise:

1. You are given a Python project with `requirements.txt` containing:

```
numpy>=1.19.0
pandas==1.2.0
scikit-learn~=0.24.0
```

The project fails with `ImportError` due to version conflicts.

2. Identify all possible version combinations that satisfy these constraints using pip's `pip-check` tool.
3. Modify `requirements.txt` to use compatible versions while keeping `pandas` fixed at 1.2.0.
4. Add a pip constraint file named `constraints.txt` to lock transitive dependencies.

Docker Multi-stage Build Exercise:

1. Create a Dockerfile that builds a Node.js application using:

```
FROM node:14 as builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
```

2. Add a second stage using `node:14-alpine` to reduce image size.
3. Configure the final image to:
 - Copy only built artifacts from `builder`
 - Expose port 3000
 - Set `NODE_ENV=production`
4. Build the image with version tag `1.0.0` and test it locally.

Semantic Versioning Exercise:

1. Given a library with current version `2.3.1`, determine the next version number for:
 - A bug fix for null pointer exceptions
 - A new feature with backward-compatible API additions
 - A breaking change that removes deprecated methods
2. Write the proper Git tag commands for each scenario.
3. Create a `.versionrc` file configuring semantic-release to:
 - Use Angular commit message format
 - Skip changelog generation for patch releases
 - Publish to npm only on main branch

2.1.3 Cross-platform gotchas (Linux, Windows, Mac)

[Cross-platform gotchas (Linux, Windows, Mac)]

File Path Handling Exercise: Write a Python function `normalize_path(path)` that converts a given file path to a platform-appropriate format. The function should:

- Replace all forward slashes with backslashes on Windows
- Replace all backslashes with forward slashes on Linux and Mac
- Handle mixed slashes in the input path
- Preserve the drive letter on Windows
- Return the normalized path as a string

Test your function with these paths:

```
"C:/Users/Public/Documents\\file.txt"
"/var/www/html\\backup/index.html"
```

Line Ending Conversion Exercise: Create a Bash script that:

- Takes a text filename as input
- Detects whether the file uses CRLF (Windows) or LF (Unix) line endings
- Converts the file to use native line endings for the current platform
- Preserves the original file permissions
- Creates a backup with `.bak` extension before conversion
- Handles binary files by skipping conversion with a warning

Use `file`, `dos2unix`, and `unix2dos` commands where appropriate.

Environment Variable Exercise: Write a C program that:

- Prints all environment variables in a platform-neutral way

- Specifically checks for `PATH`, `HOME` (Unix), and `USERPROFILE` (Windows)
- Uses `#ifdef` directives to handle platform-specific code
- Compiles without warnings on gcc, clang, and MSVC
- Implements proper memory management for Windows `_dupenv_s`

Include a `Makefile` with targets for Linux and Windows builds.

2.2 Installation Battle Plans

2.2.1 ROS workspace setup that won't break tomorrow

[ROS workspace setup that won't break tomorrow]

[Catkin Workspace Initialization] Exercise: Create a new ROS 1 Noetic catkin workspace with the following specifications:

1. Use `mkdir -p` to create the directory structure `~/catkin_ws/src`.
2. Initialize the workspace with `catkin_init_workspace` and verify the presence of `CMakeLists.txt`.
3. Build the workspace using `catkin_build` instead of `catkin_make`.
4. Source the setup file from the `devel` directory and confirm the `ROS_PACKAGE_PATH` is updated.

[Package Creation with Dependencies] Exercise: Develop a ROS package named `teleop_utils` with these requirements:

1. Use `catkin_create_pkg` with dependencies `roscpp`, `rospy`, and `geometry_msgs`.
2. Manually add `std_msgs` to the `package.xml` file as a build dependency.
3. Create a `scripts` directory for Python nodes and a `src` directory for C++ nodes.
4. Verify the package builds without errors using `catkin_build --this`.

[Workspace Overlay Debugging] Exercise: Diagnose and fix a broken ROS workspace overlay scenario:

1. Simulate the issue by sourcing two workspaces in reverse order (e.g., `source /opt/ros/noetic/setup.bash` after your workspace).
2. Identify the incorrect `ROS_PACKAGE_PATH` using `echo $ROS_PACKAGE_PATH`.
3. Resolve the conflict by re-sourcing your workspace setup file last.
4. Test the fix by running `rospack find` on your package.

2.2.2 Python virtual environments for robotics

[Python virtual environments for robotics]

[Virtual Environment Setup] Exercise: Create a Python virtual environment for a robotics project and install necessary packages.

1. Use `python -m venv` to create a virtual environment named `robotics_env`.
2. Activate the environment and upgrade `pip` to the latest version.
3. Install the packages `numpy`, `opencv-python`, and `roslibpy` with specific versions: 1.24.0, 4.5.5.64, and 0.7.0 respectively.
4. Generate a `requirements.txt` file listing all installed packages.

[Dependency Conflict Resolution] Exercise: Resolve a dependency conflict in a robotics project's virtual environment.

1. Install `pyserial==3.5` and `pymavlink==2.4.37` in a new virtual environment.
2. Attempt to install `dronekit==2.9.1`, which will fail due to dependency conflicts.
3. Use `pip check` to identify the conflicting packages.
4. Resolve the conflict by creating a constraints file or using compatible versions.

[ROS-Python Bridge] Exercise: Set up a Python virtual environment to interface with ROS (Robot Operating System).

1. Create a virtual environment named `ros_bridge` with Python 3.8.
2. Install `catkin-tools` and `rospkg` in the environment.
3. Write a Python script `talker.py` that publishes a string message to the ROS topic `/chatter`.
4. Verify the message is received by running `rostopic echo /chatter` in a separate terminal.

2.2.3 Installing Pinocchio, Drake, PyBullet without tears

[Installing Pinocchio, Drake, PyBullet without tears]

Checking Pinocchio Installation Exercise:

1. Verify Pinocchio is installed by running `import pinocchio` in Python.
2. Load the UR5 robot model from `/opt/openrobots/share/example-robot-data/robots/ur5_description/urdf/ur5.urdf`.
3. Compute the forward kinematics of the UR5 with joint angles `[0.1, -0.5, 0.3, 0.7, -0.2, 0.4]`.
4. Print the end-effector position and orientation using `SE3ToXYZQUAT`.

Drake Pendulum Simulation Exercise:

1. Install Drake using `pip install drake`.
2. Create a new Python script that imports `pydrake.all`.
3. Build a simple pendulum using `MultibodyPlant` with a 1m rod and 1kg point mass.
4. Simulate the pendulum for 5 seconds with initial angle 0.1 rad and zero velocity.
5. Plot the angle vs time using `matplotlib`.

PyBullet Collision Detection Exercise:

1. Install PyBullet via `pip install pybullet`.
2. Initialize PyBullet in GUI mode with `p.connect(p.GUI)`.
3. Load a plane and a cube at position `[0, 0, 1]` using `p.loadURDF`.
4. Simulate for 1000 steps with gravity -9.81 m/s^2 .
5. Print the final position of the cube and verify it rests on the plane.

2.2.4 Linking C++ libraries with Python bindings

[Linking C++ libraries with Python bindings]

[Building a minimal Python extension] Exercise:

1. Create a C++ file named `example.cpp` with a function `add_numbers` that takes two integers and returns their sum.
2. Write a corresponding `PyMODINIT_FUNC` module initialization function using the Python C API.
3. Compile the code into a shared library using `g++` with `-fPIC` and `-shared` flags.
4. Test the module in Python by importing it and calling `add_numbers(3, 5)`.

[Exposing a C++ class to Python] Exercise:

1. Implement a C++ class `Vector2D` with `x` and `y` members and a `magnitude` method.
2. Use `pybind11` to expose the class to Python with all members and methods.
3. Write a Python script that creates a `Vector2D` instance and prints its magnitude.
4. Handle memory management by declaring the class with `py::class_`.

[Handling NumPy arrays with pybind11] Exercise:

1. Create a C++ function `sum_array` that accepts a NumPy array via `pybind11::array_t`.
2. Include bounds checking to verify the input is a 1D array.
3. Compile the extension with NumPy support using `pybind11`'s built-in headers.
4. Write a Python test that passes both valid and invalid arrays to the function.

2.3 Development Environment Setup

2.3.1 IDE configuration for robotics workflows

[IDE configuration for robotics workflows]

[ROS Workspace Setup] Exercise:

1. Create a new ROS workspace named `robotics_ws` in your home directory
2. Initialize the workspace using `catkin_init_workspace`
3. Create a package called `sensor_processing` with dependencies `roscpp` and `sensor_msgs`
4. Build the workspace using `catkin_make`
5. Source the setup file in your shell configuration file

[VSCode Debugging] Exercise:

1. Install the C++ and ROS extensions in VSCode
2. Configure a `launch.json` file to debug a ROS node called `controller_node`
3. Set breakpoints in the callback function of a subscriber
4. Run the debug configuration and verify variable inspection works
5. Capture a screenshot of the debugger paused at a breakpoint

[URDF Visualization] Exercise:

1. Write a simple URDF file for a robot with 2 joints and 3 links
2. Configure RViz in your IDE to display the robot model
3. Add TF frames and joint state publisher GUI to the visualization
4. Create a launch file to start all visualization components
5. Verify the joints move properly when using the GUI sliders

2.3.2 Debugging tools and visualization setup

[Debugging tools and visualization setup]

[GDB Breakpoint Analysis] Exercise:

1. Write a C program named `factorial.c` that calculates the factorial of a number using recursion.
2. Compile the program with debug symbols using `gcc -g factorial.c -o factorial.`
3. Launch GDB with `gdb ./factorial` and set a breakpoint at the recursive function.
4. Run the program with argument 5 using `run 5`.

5. Use `backtrace` to inspect the call stack when the breakpoint is hit.
6. Record the values of local variables at each recursion level using `info locals`.

[Valgrind Memory Leak Detection] Exercise:

1. Create a C program named `leaky.c` that allocates memory using `malloc` but fails to free it.
2. Compile with `gcc leaky.c -o leaky` and run Valgrind using `valgrind --leak-check=yes ./leaky`.
3. Identify the line number where memory was allocated but not freed.
4. Modify the program to properly free all allocated memory.
5. Re-run Valgrind to verify no leaks are reported.

[Matplotlib Data Visualization] Exercise:

1. Write a Python script named `sensor_plot.py` using Matplotlib.
2. Generate synthetic sensor data with timestamps and random values between 0-100.
3. Plot the data as a line graph with proper labels for X-axis (time) and Y-axis (value).
4. Add a horizontal dashed line at `y=50` representing a threshold.
5. Save the plot as `sensor_output.png` with 300 DPI resolution.

2.3.3 Testing your installation before you need it

[Testing your installation before you need it]

[Verifying Python Environment] Exercise:

1. Open a terminal and run `python --version` to confirm Python 3.7 or later is installed.
2. Create a file named `test_env.py` with the following content:

```
import sys
print(f"Python version: {sys.version}")
print(f"Path: {sys.path}")
```

3. Execute the script and verify no errors occur in the output.
4. Install the `numpy` package using `pip install numpy` and repeat step 3.

[Checking Compiler Toolchain] Exercise:

1. Create a C file named `hello_world.c` with the following content:

```
#include
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

2. Compile it using `gcc hello_world.c -o hello`.
3. Run the executable with `./hello` and verify the output.
4. Repeat the process with the `-Wall` flag and check for warnings.

[Testing Network Connectivity] Exercise:

1. Use `ping 8.8.8.8` to test basic internet connectivity.
2. Run `nslookup example.com` to verify DNS resolution.
3. Create a Python script using the `requests` library to fetch `https://example.com`.
4. Modify the script to check the HTTP status code equals 200.

Chapter 3

Forward Kinematics: From Joint Angles to "Where Is My Robot?"

3.1 Single Manipulator Basics

3.1.1 Forward kinematics in Pinocchio: Copy-paste examples

[Forward kinematics in Pinocchio: Copy-paste examples]

[Single-Link Transformation] Exercise:

1. Create a 1-DOF robot arm using Pinocchio with a revolute joint along the Z-axis.
2. Place the joint at position $(0.5, 0, 0)$ relative to the world frame.
3. Use `pinocchio::forwardKinematics` to compute the end-effector position when the joint angle is 45 degrees.
4. Verify the result by manually calculating the expected position using homogeneous transformation matrices.

[UR5 End-Effector Position] Exercise:

1. Load the UR5 model using `pinocchio::buildModels.ur5`.
2. Set all joint angles to zero except the shoulder joint (joint 2), which should be at 30 degrees.
3. Compute the end-effector position using forward kinematics.
4. Compare the result with the theoretical position using the UR5's documented link lengths (shoulder-to-elbow: 0.425m, elbow-to-wrist: 0.392m).

[Humanoid Leg Forward Kinematics] Exercise:

1. Create a simplified humanoid leg model with 3 joints: hip (revolute Y), knee (revolute Y), and ankle (revolute Y).
2. Set the link lengths as: hip-to-knee = 0.5m, knee-to-ankle = 0.4m, ankle-to-foot = 0.1m.
3. Implement forward kinematics for joint angles: hip = 20° , knee = -30° , ankle = 10° .
4. Compute the foot position and compare with a manual calculation using the Denavit-Hartenberg convention.

3.1.2 Using Drake's pydrake: When you need more control

[Using Drake's pydrake: When you need more control]

[Inverse Kinematics with Position Constraints] Exercise: Implement an inverse kinematics (IK) problem for a 7-DOF robotic arm using pydrake. The goal is to position the end-effector at a target location while avoiding joint limits. Follow these steps:

1. Load the IIWA arm URDF using `FindResourceOrThrow` and create a `MultibodyPlant`.
2. Add a `PositionConstraint` to enforce the end-effector's position at `target_xyz`.
3. Add `BoundingBoxConstraint` for each joint to limit its range to `[-pi, pi]`.
4. Solve the IK problem using `Solve` from `MathematicalProgram`.
5. Print the resulting joint angles and verify the end-effector error is below $1e-3$.

[Trajectory Optimization for a Cart-Pole] Exercise: Design a swing-up trajectory for a cart-pole system using direct transcription. The cart must return to its starting position after 3 seconds. Follow these steps:

1. Create a `MultibodyPlant` of the cart-pole system from Drake's examples.
2. Initialize a `DirectTranscription` with 50 time samples over 3 seconds.
3. Add a cost term to minimize $(\text{theta} - \pi)^2$ where `theta` is the pole angle.
4. Enforce zero final velocity for both cart and pole via `AddConstraint`.
5. Solve with `SolveSQP` and plot the state trajectory using `matplotlib`.

[Custom LeafSystem for Signal Processing] Exercise: Implement a real-time moving average filter as a pydrake `LeafSystem`. The filter should average inputs over a 0.5-second sliding window. Follow these steps:

1. Derive from `LeafSystem` and declare one input port for scalar signals.
2. Override `DoCalcVectorOutput` to maintain a buffer of recent values.
3. Use `get_time` and a deque to discard samples older than 0.5 seconds.
4. Test your system by connecting it to a `Sine` source in a `DiagramBuilder`.
5. Verify the output smooths the sine wave while introducing less than 0.1s delay.

3.1.3 Coordinate frames: The source of 80 percent of your bugs

[Coordinate frames: The source of 80

Frame Transformation for Robot Arm Exercise: A 2-DOF robotic arm has joint angles θ_1 and θ_2 with link lengths $l_1 = 0.5\text{ m}$ and $l_2 = 0.3\text{ m}$.

1. Derive the transformation matrix from the base frame to the end-effector frame.
2. Given $\theta_1 = 30^\circ$ and $\theta_2 = 45^\circ$, compute the end-effector position in the base frame.

3. A camera detects an object at $(0.6, 0.2)$ in its frame, which is rotated by 20° relative to the base frame. Convert this position to the base frame.

GPS to Local ENU Conversion Exercise: A drone receives GPS coordinates $(40.7128^\circ N, 74.0060^\circ W, 50\text{ m})$ and needs to convert to a local East-North-Up (ENU) frame centered at $(40.7125^\circ N, 74.0055^\circ W, 0\text{ m})$.

1. Calculate the geodetic to ECEF (Earth-Centered Earth-Fixed) conversion for both points.
2. Compute the ENU transformation matrix for the local frame.
3. Convert the drone's position to ENU coordinates, assuming WGS84 ellipsoid parameters.

Sensor Fusion Frame Alignment Exercise: An IMU reports linear acceleration $\mathbf{a}_{imu} = [0.2, -0.1, 9.8]\text{ m/s}^2$ in its body frame, which is mounted with a 10° rotation about the vehicle's x-axis.

1. Write the rotation matrix between IMU and vehicle frames.
2. Transform the acceleration measurement to the vehicle frame.
3. The vehicle is pitched upward by 15° . Transform the acceleration to the world frame.

3.2 Multi-body and Complex Chains

3.2.1 Computing poses in MoveIt: The ROS way

[Computing poses in MoveIt: The ROS way]

[Forward Kinematics with MoveGroup] Exercise:

1. Launch a ROS node with a `MoveGroupInterface` object for the `panda_arm` planning group.
2. Set a joint state goal for the Panda arm: `joint1 = 0.5, joint2 = -0.3, joint3 = 0.7, joint4 = -1.2, joint5 = 0.5, joint6 = 0.8, joint7 = 0.0` (all values in radians).
3. Compute the end-effector pose using `getCurrentPose()` after setting the joint goal.
4. Print the resulting position (x,y,z) and quaternion orientation (x,y,z,w) to the terminal.

[Pose Transformation with TF2] Exercise:

1. Initialize a TF2 buffer and listener in a ROS node.
2. Broadcast a static transform from `base_link` to `camera_frame` with translation $(0.5, 0.1, 0.3)$ and rotation $(0, 0, 0.3826834, 0.9238795)$ as a quaternion.
3. Create a pose stamped message for a point at $(0.2, 0.0, 0.1)$ relative to `camera_frame` with identity orientation.
4. Transform this pose to `base_link` frame using `tf2_ros::Buffer::transform()`.
5. Verify the transformed pose matches the expected manual calculation.

[Collision-Aware Pose Sampling] Exercise:

1. Load the Panda robot URDF into a `moveit::core::RobotModel`.
2. Create a planning scene with a 20cm cubic obstacle centered at (0.4, 0.0, 0.5) in `base_link` frame.
3. Sample 100 valid end-effector poses in a workspace box `x=[0.3,0.6]`, `y=[-0.3,0.3]`, `z=[0.1,0.7]`.
4. Filter poses that collide with the obstacle using `PlanningScene::isStateValid()`.
5. Visualize collision-free poses as markers in RViz using `visualization_msgs::MarkerArray`.

3.2.2 Forward kinematics in PyBullet simulation

[Forward kinematics in PyBullet simulation]

[Joint Position Calculation] Exercise: Given a 2R planar robotic arm in PyBullet with link lengths $l_1 = 0.5$ m and $l_2 = 0.3$ m:

1. Compute the end-effector position analytically for joint angles $\theta_1 = 45^\circ$ and $\theta_2 = 30^\circ$
2. Implement the forward kinematics calculation in Python using PyBullet's `getJointState()` function
3. Verify your result by comparing with PyBullet's `getLinkState()` output
4. Plot the arm configuration using `matplotlib` with labeled joints and links

[URDF Forward Kinematics] Exercise: For a UR5 robot loaded in PyBullet via `loadURDF()`:

1. Identify all revolute joints in the URDF model using `getJointInfo()`
2. Set joint angles to `[0.1, -0.5, 0.3, 1.2, -1.0, 0.8]` radians using `setJointMotorControl2()`
3. Compute the end-effector pose using the product of exponentials method
4. Extract the end-effector position from PyBullet using `getLinkState(linkIndex=5)`
5. Calculate the position error between your computation and PyBullet's result

[Visualization and Validation] Exercise: For a custom 3-DOF robotic arm in PyBullet:

1. Create a simple URDF file defining three revolute joints with tags
2. Load the model and set up a basic simulation environment with gravity
3. Implement forward kinematics using Denavit-Hartenberg parameters
4. Visualize the computed frames using `pybullet.addUserDebugLine()`
5. Compare frame positions with PyBullet's debug visualization mode

3.2.3 Handling branched kinematic trees

[Handling branched kinematic trees]

Forward kinematics for a robotic arm Exercise:

1. A 3-DOF robotic arm has the following Denavit-Hartenberg parameters for each joint:

```
Joint 1: theta = q1, d = 0.5, a = 0.2, alpha = pi/2
Joint 2: theta = q2, d = 0, a = 1.0, alpha = 0
Joint 3: theta = q3, d = 0, a = 0.8, alpha = 0
```

2. Compute the homogeneous transformation matrix from the base to the end-effector.
3. Implement this transformation in Python using `numpy`, with q_1 , q_2 , and q_3 as inputs.
4. Calculate the end-effector position for $q_1 = 0.1$, $q_2 = 0.5$, $q_3 = -0.3$ (radians).

Branching manipulator kinematics Exercise:

1. A mobile robot has two parallel 2-DOF arms mounted on its base.
2. The first arm has link lengths $l_1 = 0.4$ and $l_2 = 0.3$.
3. The second arm has link lengths $l_1 = 0.35$ and $l_2 = 0.25$.
4. Derive the forward kinematics for both arms relative to the robot's base frame.
5. Write a MATLAB function that takes joint angles for both arms and returns their end-effector positions.
6. Compute the workspace overlap area where both end-effectors can reach simultaneously.

Inverse kinematics for tree structures Exercise:

1. A 4-DOF branched manipulator splits at joint 2 into two 2-DOF chains.
2. The left branch has link lengths $a_1 = 0.6$, $a_2 = 0.4$.
3. The right branch has link lengths $b_1 = 0.5$, $b_2 = 0.3$.
4. Given desired positions $p_{\text{left}} = [0.7, 0.2]$ and $p_{\text{right}} = [0.4, 0.5]$ for both end-effectors:
5. Formulate the inverse kinematics problem as an optimization task.
6. Implement a numerical solution using Python's `scipy.optimize`.
7. Handle the joint limit constraints $-\pi/2 \leq q_i \leq \pi/2$ during optimization.

3.3 Common Gotchas and Red Flags

3.3.1 When your end-effector is in the wrong universe

[When your end-effector is in the wrong universe]

Forward Kinematics Calibration Exercise: Exercise:

1. Given a 2R planar robot with link lengths $l_1 = 0.5\text{m}$ and $l_2 = 0.3\text{m}$, compute the end-effector position when joint angles are $\theta_1 = 45^\circ$ and $\theta_2 = 30^\circ$.
2. The actual measured end-effector position is $x = 0.55\text{m}$, $y = 0.65\text{m}$. Calculate the Euclidean distance error from the theoretical position.
3. Propose two possible mechanical calibration adjustments to reduce this error.

Reference Frame Transformation Exercise: Exercise:

1. A robot's base frame $\{B\}$ is offset from world frame $\{W\}$ by $(0.2\text{m}, -0.1\text{m}, 0.0\text{m})$ with no rotation. The end-effector position in $\{B\}$ is $[0.4, 0.3, 0.0]^T$. Compute its world coordinates.
2. The desired position in $\{W\}$ is $[0.7, 0.1, 0.0]^T$. Determine if the end-effector is in the correct workspace quadrant.
3. Write a Python function using `numpy` to transform any point from $\{B\}$ to $\{W\}$ given the translation offset.

```
import numpy as np
def transform_B_to_W(point_B, offset_W):
    # Your implementation here
```

Sensor Fusion for Position Correction Exercise: Exercise:

1. An end-effector has encoder readings giving position $p_{\text{enc}} = [1.02, 0.98]\text{m}$ and a camera system reports $p_{\text{cam}} = [0.95, 1.03]\text{m}$. Compute the weighted average using weights $w_{\text{enc}} = 0.7$ and $w_{\text{cam}} = 0.3$.
2. The standard deviations are $\sigma_{\text{enc}} = 0.05\text{m}$ and $\sigma_{\text{cam}} = 0.1\text{m}$. Recompute using variance-weighted fusion.
3. Implement a Kalman filter prediction step in Python assuming constant velocity:

```
def kalman_predict(x, P, F, Q):
    # Your implementation here
```

3.3.2 Joint limits and what happens when you ignore them

[Joint limits and what happens when you ignore them]

Forward kinematics with joint limits Exercise: A 2R planar robot has link lengths $l_1 = 0.5\text{ m}$ and $l_2 = 0.3\text{ m}$ with joint limits:

- Joint 1: $q_{1_min} = -90^\circ$, $q_{1_max} = 90^\circ$

- Joint 2: $q2_min = 0 \text{ deg}$, $q2_max = 120 \text{ deg}$

Compute the Cartesian position for these joint configurations and state if they violate limits:

1. $q = [30 \text{ deg}, 45 \text{ deg}]$
2. $q = [-95 \text{ deg}, 10 \text{ deg}]$
3. $q = [80 \text{ deg}, 130 \text{ deg}]$

Joint limit violation effects Exercise: A robotic arm with joint torque sensors shows these readings during motion:

Time (s)	Torque (Nm)
0.0	5.2
0.5	7.8
1.0	15.3 (joint limit hit)
1.5	22.1
2.0	18.6

- Plot the torque vs time data using any tool
- Identify the moment when joint limits were exceeded
- Calculate the average torque increase rate before/after limit contact

Software safety check implementation Exercise: Write a Python function to validate joint positions against limits:

```
def check_limits(q, q_min, q_max):
    # Your code here
```

1. The function should return `True` if all joints are within limits
2. Include a check for NaN or infinity values
3. Test with $q = [1.0, -0.5, 3.14]$, $q_min = [-1.0, -1.0, 0.0]$, $q_max = [1.0, 1.0, 2\pi]$

3.3.3 Performance: When FK becomes a bottleneck

[Performance: When FK becomes a bottleneck]

[Indexing Foreign Keys] Exercise: A database table `orders` has a foreign key `customer_id` referencing `customers.id`. The query `SELECT * FROM orders WHERE customer_id = 123` runs slowly.

1. Write the SQL command to add an index that would optimize this query.
2. Explain why this index improves performance for the given query.
3. List two potential downsides of adding this index to the `orders` table.

[Batch Processing for FK Checks] Exercise: A bulk import operation inserts 10,000 rows into a table with foreign key constraints.

1. Write a PostgreSQL command to temporarily disable FK checks during the import.
2. Describe how to re-enable FK checks after the import completes.
3. Explain why disabling FK checks might cause data integrity issues if not handled properly.

[Denormalizing for Performance] Exercise: A social media platform has tables `users` and `posts` with a FK `posts.user_id`. The query `SELECT u.username, p.* FROM posts p JOIN users u ON p.user_id = u.id` is too slow.

1. Propose a denormalized schema that would eliminate the need for this join.
2. Write the SQL to create your proposed denormalized table.
3. Identify one business scenario where this denormalization would cause problems.

Chapter 4

Inverse Kinematics: Making Robots Go Where You Want

4.1 The Reality of IK Solvers

4.1.1 Why there's no "perfect" IK solution

[Why there's no "perfect" IK solution]

Analytical IK for 2R Planar Arm Exercise: Given a 2R planar robotic arm with link lengths $l_1 = 0.5\text{m}$ and $l_2 = 0.3\text{m}$:

1. Derive the forward kinematics equations for end-effector position (x,y).
2. Calculate all possible joint angles (θ_1, θ_2) for the end-effector at (0.6m, 0.2m).
3. Plot both configurations using

```
import matplotlib.pyplot as plt
```

showing the elbow-up and elbow-down solutions.

Jacobian Singularity Detection Exercise: For a 3DOF robotic arm with the Jacobian matrix:

```
J = [[-l1*s1 - l2*s12, -l2*s12, 0],  
      [ l1*c1 + l2*c12,  l2*c12, 0],  
      [ 0, 0, 1]]
```

where $s1 = \sin(\theta_1)$, $c12 = \cos(\theta_1 + \theta_2)$, etc.:

1. Compute the determinant symbolically.
2. Identify all joint angle combinations that make the determinant zero.
3. Write a Python function `is_singular(theta1, theta2)` returning a boolean.

Iterative IK Implementation Exercise: Implement CCD (Cyclic Coordinate Descent) for a 3-link arm:

1. Initialize joint angles $\theta_{\text{etas}} = [0, 0, 0]$ with all links length 1m.

2. Write the CCD update step for joint i using `atan2`.
3. Stop when error $e = |\text{target} - \text{current}|$ is below 0.01m.
4. Test on target (2.0m, 1.5m) and plot each iteration's arm configuration.

4.1.2 Numerical vs analytic: What the libraries actually do

[Numerical vs analytic: What the libraries actually do]

ODE Solver Implementation Exercise: Implement Euler's method to solve the first-order ODE:

$$dy/dt = -k*y, \text{ with } y(0) = y_0$$

1. Write a Python function `euler_solve(k, y0, t_end, n_steps)` that returns time array and solution array
2. Compare results with exact solution $y(t) = y_0 \exp(-k*t)$ at $t=1.0$ for $k=0.5$, $y_0=1.0$
3. Plot both solutions for $n_steps=10$ and $n_steps=100$ on the same axes using `matplotlib`
4. Calculate the absolute error at $t=1.0$ for both step counts

Matrix Library Benchmark Exercise: Investigate numerical precision in matrix operations:

1. Create a 10x10 Hilbert matrix H using `scipy.linalg.hilbert(10)`
2. Compute its inverse H_inv using `numpy.linalg.inv()`
3. Multiply H and H_inv to get identity matrix I
4. Calculate the Frobenius norm `np.linalg.norm(I - np.eye(10), 'fro')`
5. Repeat using `scipy.linalg.invhilbert(10)` as exact inverse and compare norms

Special Function Accuracy Exercise: Evaluate numerical implementations of Bessel functions:

1. Using `scipy.special.jv(1, x)`, compute $J_1(x)$ for $x=5.0$
2. Compare with the first 10 terms of the series expansion:

$$J_1(x) = \sum_{m=0}^9 \left[\frac{(-1)^m}{m! \cdot \Gamma(m+2)} \right] * (x/2)^{(2m+1)}$$

3. Plot the absolute difference between the two methods for x in $[0.1, 10.0]$
4. At what x value does the relative error exceed $1e-6$?

4.1.3 Convergence, failure modes, and what to do about them

[Convergence, failure modes, and what to do about them]

[Gradient Descent Divergence] Exercise: A gradient descent implementation for minimizing $f(x) = x^4 - 3x^3 + 2$ diverges when using a fixed step size $\alpha = 1.2$.

1. Identify the mathematical condition that causes this divergence.
2. Modify the step size to $\alpha = 0.1$ and perform three iterations starting at $x_0 = 2.5$.
3. Rewrite the update rule using momentum $\beta = 0.9$ and show the first two iterations.

[Newton's Method Failure] Exercise: The function $f(x) = |x|^{1.5}$ causes Newton's method to fail when $x_0 = -1$.

1. Compute the first Newton iteration step explicitly and identify the failure point.
2. Propose a safeguarded update rule that prevents divergence.
3. Implement your solution in Python using

```
def newton_safeguarded(x0, max_iter=10):
    # Your code here
```

[Linear System Solver Diagnostics] Exercise: A conjugate gradient solver for $Ax = b$ stalls with residual $\|r_k\| = 10^{-6}$ after 100 iterations.

1. List three possible numerical causes for this stagnation.
2. Write MATLAB code to compute the condition number of A using

```
condest(A)
```

3. Propose a preconditioning strategy and show the modified system setup.

4.2 Practical IK Workflows

4.2.1 IK in MoveIt: The easy button (when it works)

[IK in MoveIt: The easy button (when it works)]

[Basic IK Query] Exercise: Given a robotic arm with a MoveIt configuration, write a Python script to:

1. Initialize a `MoveGroupInterface` for the arm's planning group
2. Set a target pose for the end-effector with position (0.5, 0.2, 0.3) and identity orientation
3. Call `compute_ik()` to solve for joint angles
4. Print the solution or an error message if no solution exists

[IK with Constraints] Exercise: Extend the basic IK solution to:

1. Add a path constraint limiting the end-effector orientation to remain within 0.1 radians of the identity quaternion
2. Set the planning time to 5 seconds using `set_planning_time()`
3. Visualize the solution in RViz using `display_trajectory()`
4. Handle cases where the constrained solution fails

[IK for Multiple Waypoints] Exercise: Create a trajectory using IK solutions for three waypoints:

1. Define waypoints at (0.5, 0.2, 0.3), (0.6, 0.1, 0.4), and (0.4, 0.3, 0.2)
2. Compute IK solutions for each waypoint with default orientation
3. Use `compute_cartesian_path()` to generate a continuous trajectory
4. Check for collisions using `check_path_validity()`
5. Execute the trajectory if valid, otherwise print an error

4.2.2 Pinocchio IK solvers: More control, more complexity

[Pinocchio IK solvers: More control, more complexity]

[Joint Limits and IK Feasibility] Exercise: Given a robotic arm with 6 revolute joints, each with limits q_{\min} and q_{\max} :

1. Write a Python function using Pinocchio to check if a candidate joint configuration q respects all joint limits.
2. Modify the function to return a boolean mask indicating which joints violate their limits.
3. Implement a clamping function that forces out-of-bounds values to their nearest valid limit.

[IK Solver Tuning] Exercise: For a 7-DOF manipulator using Pinocchio's `levenbergMarquardt` solver:

1. Write a script to solve IK for a target end-effector pose `SE3_target`.
2. Experiment with damping factor `lambda` values (0.01, 0.1, 1.0) and report convergence behavior.
3. Add joint limit avoidance by modifying the cost function with a quadratic penalty term.

[Multiple Task Prioritization] Exercise: For a humanoid robot with both hand and foot IK targets:

1. Formulate two separate task Jacobians J_{hand} and J_{foot} using Pinocchio.
2. Implement a weighted task-priority IK solver using null-space projection.
3. Visualize the resulting motion when prioritizing hand positioning over foot placement.

4.2.3 Drake IK: Optimization-based approaches

[Drake IK: Optimization-based approaches]

[Inverse Kinematics with Position Constraints] Exercise: Given a 7-DOF robotic arm modeled in Drake, solve the inverse kinematics (IK) problem for a target end-effector position `p_target`. Assume the robot is modeled as `MultibodyPlant`. Perform the following steps:

1. Construct a `MathematicalProgram` to minimize the squared Euclidean distance between the end-effector position and `p_target`.
2. Add joint limit constraints using `prog.AddBoundingBoxConstraint`.
3. Use `Solve` to compute the joint angles `q_sol`.
4. Verify the solution by computing the forward kinematics of `q_sol` and checking the error against `p_target`.

[IK with Orientation Constraints] Exercise: Extend the previous exercise to include orientation constraints. The goal is to match both position `p_target` and orientation `R_target` (as a rotation matrix). Perform the following:

1. Define a cost term for orientation error using the Frobenius norm of $R(q) - R_{\text{target}}$, where $R(q)$ is the end-effector rotation matrix.
2. Combine position and orientation costs into a weighted sum using `prog.AddCost`.
3. Solve the IK problem and validate the solution by comparing $R(q_{\text{sol}})$ to R_{target} .

[Collision-Aware IK] Exercise: Solve an IK problem while avoiding collisions between the robot and a box obstacle defined by its center `box_center` and size `box_size`. Perform the following:

1. Use `SceneGraphInspector` to check for collisions between the robot and the box.
2. Add a signed-distance constraint using `prog.AddConstraint` to ensure all robot geometry stays outside the box.
3. Solve the IK problem and visualize the result in MeshCat to confirm no collisions exist.

4.2.4 PyBullet IK: Quick and dirty for simulation

[PyBullet IK: Quick and dirty for simulation]

Inverse Kinematics for a UR5 Robot Exercise:

1. Load the UR5 robot model in PyBullet using `p.loadURDF()` with the `flags=p.URDF_USE_INERTIA_FROM_FILE` option.
2. Set the end-effector link index to 7 (`wrist_3_link`) and define a target position `target_pos = [0.5, 0.2, 0.8]` in world coordinates.
3. Use `p.calculateInverseKinematics()` with 100 iterations and residual threshold of 0.001 to solve for joint angles.

4. Verify the solution by setting the robot's joint states and visually checking end-effector alignment in the PyBullet GUI.

IK with Null Space Control Exercise:

1. Load the KUKA LBR iiwa robot model and set the end-effector to link 6.
2. Define a target orientation `target_orn = p.getQuaternionFromEuler([0, 1.57, 0])`.
3. Solve IK using the null space parameters to prefer joint configuration `null_space_bias = [0, 0.5, 0, 0.5, 0, 0.5, 0]`.
4. Compare the results with and without null space control by plotting joint angles using `matplotlib`.

IK for Redundant Manipulator Exercise:

1. Implement a 7-DOF Panda arm simulation with PyBullet.
2. Define a straight-line Cartesian path of 5 waypoints for the end-effector.
3. Use `p.calculateInverseKinematics()` with `damping=0.05` to solve for each waypoint.
4. Animate the motion and check for discontinuities or jerky movements between waypoints.

4.3 When IK Goes Wrong

4.3.1 Detecting bad solutions before they break things

[Detecting bad solutions before they break things]

[Threshold-based anomaly detection] Exercise: A temperature sensor in a chemical reactor outputs values every 5 seconds. Implement a Python function that detects anomalies using a simple threshold method:

1. Write a function `check_temperature(value, lower_threshold, upper_threshold)` that returns `True` if the value is outside the specified range
2. The function should log warnings using `print()` when values exceed thresholds
3. Add a 10% buffer zone where values within 10% of thresholds trigger warnings but don't count as anomalies
4. Test your function with this input sequence: `[85, 90, 105, 110, 200, 75, 91]` using thresholds 90 and 110

[CRC checksum verification] Exercise: A communication protocol uses 8-bit CRC checksums to detect transmission errors:

1. Implement a Python function `compute_crc(data)` that calculates the CRC-8 checksum using polynomial 0x07

2. Create a second function `verify_packet(packet)` where packet format is `[payload, crc]`
3. The verification function should return `True` if the computed CRC matches the packet's CRC
4. Test with these packets: `[[0x01, 0x02], 0x15]` (valid) and `[[0xFF, 0x00], 0x00]` (invalid)

[Heartbeat monitoring] Exercise: A distributed system uses heartbeat messages between nodes:

1. Design a Python class `HeartbeatMonitor` with methods `beat()` and `is_alive(timeout=5)`
2. The class should track the last received heartbeat timestamp internally
3. Method `is_alive()` should return `False` if no heartbeat arrived within timeout seconds
4. Add a property `time_since_last_beat` that returns seconds since last heartbeat
5. Simulate a failing node by not calling `beat()` for 6 seconds and verify detection

4.3.2 Multiple solutions: Which one to pick?

[Multiple solutions: Which one to pick?]

[Thermal Efficiency Comparison] Exercise:

1. A heat engine operates between two reservoirs at 800 K and 300 K. Calculate its maximum theoretical efficiency.
2. The same engine is modified with an intermediate reservoir at 500 K. Compute the new maximum efficiency.
3. Compare both results and explain which configuration is preferable for power generation.

[Circuit Analysis Tradeoffs] Exercise:

1. Design a voltage divider circuit using two resistors to produce 3.3 V from a 5 V source.
2. Repeat the design using three resistors in series to achieve the same output.
3. Calculate the power dissipation for both designs when loaded with $R_{load} = 1k\Omega$.
4. State which design is more energy-efficient and justify your choice.

[Database Query Optimization] Exercise:

1. Write a SQL query to find all customers who made purchases over \$100 in January 2023:

```
SELECT * FROM customers
JOIN orders ON customers.id = orders.customer_id
WHERE orders.amount > 100 AND orders.date BETWEEN '2023-01-01' AND '
2023-01-31';
```

2. Rewrite the query using a subquery instead of a JOIN.
3. Execute both queries on a sample database with `EXPLAIN ANALYZE` and compare their execution times.
4. Recommend which version should be used in production and why.

4.3.3 Seed points and why they matter more than you think

[Seed points and why they matter more than you think]

Visualizing seed point clustering Exercise:

1. Generate 100 random 2D seed points with coordinates between 0 and 1 using `numpy.random.rand()`.
2. Plot these points using `matplotlib.pyplot.scatter()` with default parameters.
3. Calculate and display the Euclidean distance matrix between all points using `scipy.spatial.distance.cdist()`.
4. Identify the 5 closest seed point pairs and highlight them in red on your plot.

Seed point initialization for k-means Exercise:

1. Load the Iris dataset using `sklearn.datasets.load_iris()` and extract the first two features.
2. Implement k-means clustering with `k=3` using two initialization methods: random seeds and `k-means++`.
3. Compare the final inertia values for both methods after 10 runs each.
4. Plot the resulting clusters for the best run of each method side by side.

Seed point sensitivity analysis Exercise:

1. Create a synthetic dataset of 300 points forming three clear Gaussian clusters using `sklearn.datasets.make_blobs()`.
2. Run k-means clustering 50 times with random seed initialization, recording the final inertia each time.
3. Plot a histogram of the inertia values and calculate the standard deviation.
4. Identify the worst and best clustering results by inertia and visualize their cluster assignments.

Chapter 5

Jacobians: The Bridge Between Joint and Task Space

5.1 Computing Jacobians Without Crying

5.1.1 Pinocchio Jacobian API: Fast and reliable

[Pinocchio Jacobian API: Fast and reliable]

Frame Jacobian Computation Exercise: Exercise:

1. Using Pinocchio's `computeFrameJacobian`, compute the Jacobian of the right foot frame for a humanoid robot in the `LOCAL_WORLD_ALIGNED` representation.
2. The robot model is loaded from `/path/to/urdf`. Assume joint configuration q is a vector of all zeros except for a 0.5 radian knee bend.
3. Verify the Jacobian's translational block by comparing against finite differences of `forwardKinematics` results with $\Delta q = 1e^{-6}$.

Jacobian Time Derivative Exercise: Exercise:

1. Implement a function that computes \dot{J} using `computeJointJacobiansTimeVariation` for a 7-DOF robotic arm.
2. The input should be joint configuration q and velocity v (both as NumPy arrays).
3. Benchmark against the analytical derivative of `computeFrameJacobian` using `scipy.optimize.check_grad`.

Operational Space Control Exercise: Exercise:

1. Design a Cartesian PD controller using the Jacobian pseudoinverse to track a circular end-effector trajectory.
2. The system has a 6D task-space error defined as $[p_err, \theta_err]$.
3. Use `pinocchio::Data::J` and `pinocchio::Data::M` to implement the dynamically consistent pseudoinverse.
4. Log the RMS error over 100 time steps with $\Delta t = 0.01s$.

5.1.2 Drake Jacobian utilities: When you need derivatives too

[Drake Jacobian utilities: When you need derivatives too]

[Jacobian of a planar rotation] Exercise:

1. Implement a function `rotation_jacobian(theta)` in Python that computes the Jacobian of the 2D rotation matrix $R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$ with respect to θ .
2. Verify your implementation by numerically differentiating $R(\theta)$ using central differences with $\Delta\theta = 10^{-6}$ at $\theta = \pi/4$.
3. Plot the Frobenius norm of the difference between your analytical Jacobian and the numerical approximation for $\theta \in [0, 2\pi]$.

[Manipulator Jacobian validation] Exercise:

1. Given a 2R planar manipulator with link lengths $l_1 = 0.5$, $l_2 = 0.3$, implement its geometric Jacobian $J(q)$ where $q = [q_1, q_2]$ are joint angles.
2. Compute the end-effector velocity for $q = [0.1, 0.2]$ and $\dot{q} = [0.5, -0.3]$ (joint velocities).
3. Check your result by finite-differencing the forward kinematics over $\Delta t = 10^{-6}$ with the same joint configuration.

[Auto-diff for Jacobian computation] Exercise:

1. Use Drake's `AutoDiffXd` to compute the Jacobian of the scalar function $f(x, y) = x^2 \sin(y) - ye^x$ at $(x, y) = (1, 2)$.
2. Compare the result with the analytical Jacobian computed by hand.
3. Create a table showing the absolute error between the auto-diff and analytical results for 5 random points in $[-1, 1] \times [-\pi, \pi]$.

5.1.3 Numerical vs analytical: Speed vs accuracy trade-offs

[Numerical vs analytical: Speed vs accuracy trade-offs]

[Finite difference approximation] Exercise:

1. Implement a Python function `finite_diff(f, x, h)` that computes the forward difference approximation of the derivative of f at point x with step size h .
2. Calculate the absolute error between your approximation and the analytical derivative for $f(x) = \exp(2x)$ at $x = 1$ with $h = 0.1$.
3. Repeat the calculation for $h = 1e-6$ and explain why the error changes.
4. Plot the error vs h for $h = 10^{-k}$ where k ranges from 1 to 15.

[ODE solver comparison] Exercise:

1. Solve the differential equation $dy/dt = -2y$ with $y(0) = 1$ analytically.

2. Implement Euler's method in Python to solve the same equation numerically over $t = [0, 5]$ with $dt = 0.1$.
3. Implement the 4th-order Runge-Kutta method for the same problem.
4. Compare the maximum absolute error and computation time for both methods against the analytical solution.

[Matrix inversion methods] Exercise:

1. Generate a random 5×5 positive definite matrix A and vector b .
2. Solve $Ax = b$ using Python's `numpy.linalg.solve` and record the time.
3. Implement Gaussian elimination without pivoting and solve the same system.
4. Compare the residual $\|Ax - b\|_2$ and computation time for both methods.
4. Repeat with a Hilbert matrix of size 10×10 and observe the differences.

5.2 Using Jacobians in Real Applications

5.2.1 Joint velocity to end-effector velocity mapping

[Joint velocity to end-effector velocity mapping]

[Jacobian matrix computation] Exercise:

1. A 2R planar robotic arm has link lengths $l_1 = 0.5$ m and $l_2 = 0.3$ m. Compute the analytical Jacobian matrix when joint angles are $\theta_1 = \pi/4$ and $\theta_2 = \pi/6$.
2. Implement a Python function using `numpy` to compute this Jacobian numerically:

```
def compute_jacobian(theta1, theta2, l1, l2):
    # Your implementation here
```

3. Verify that your numerical result matches the analytical solution within $1e-6$ tolerance.

[Singularity analysis] Exercise:

1. For the same 2R arm, identify all joint configurations where the Jacobian becomes singular.
2. Calculate the end-effector velocity vector when $\theta_1 = 0$, $\theta_2 = 0$ and joint velocities are $\dot{q} = [1, 1]$ rad/s.
3. Explain why this results in a degenerate motion in Cartesian space.

[Velocity transformation] Exercise:

1. Given a 3-DOF robotic arm with Jacobian $J = \begin{bmatrix} -0.2 & 0.3 & 0 \\ 0.4 & -0.1 & 0.5 \end{bmatrix}$ at some configuration. Compute the end-effector velocity when $\dot{q} = [0.1, -0.2, 0.3]$ rad/s.
2. The end-effector requires a velocity of $[0.5, -0.2]$ m/s. Find the required joint velocities using the pseudoinverse of J .
3. Validate your solution by computing the resulting end-effector velocity.

5.2.2 Force/torque mapping: When robots push back

[Force/torque mapping: When robots push back]

Static Force Analysis Exercise:

1. A robotic arm with a 3D force/torque sensor at its wrist is pushing against a rigid wall. The sensor measures $\vec{F} = [5.2, -1.8, 3.4]^T$ N and $\vec{\tau} = [0.4, 0.6, -0.2]^T$ Nm. The contact point is at $\vec{r} = [0.1, 0.05, 0]^T$ m from the sensor origin. Verify that $\vec{\tau} = \vec{r} \times \vec{F}$ holds within a 5% tolerance.
2. Implement a Python function `check_wrench_consistency(F, tau, r)` that returns `True` if the torque matches the cross product within tolerance. Use this to verify the measurements from task 1.
3. The same arm now pushes at $\vec{r} = [0, 0.15, 0.03]^T$ m with $\vec{F} = [-2.1, 4.7, 1.2]^T$ N. Calculate the expected torque and compare with a measured $\vec{\tau} = [0.25, 0.06, 0.315]^T$ Nm.

Compliance Control Implementation Exercise:

1. Write a ROS node that subscribes to `/ft_sensor` (`geometry_msgs/WrenchStamped`) and publishes desired joint velocities on `/joint_vel_cmds`. The node should implement $\dot{q} = J^{-1}K_f(F_{des} - F_{meas})$ where K_f is a diagonal gain matrix.
2. Extend the node to handle torque limits: if any joint torque from $\tau = J^T F$ exceeds 10 Nm, scale down all velocities proportionally.
3. Test your node with $F_{des} = [0, 0, -10]^T$ N and $K_f = \text{diag}(0.1, 0.1, 0.2)$. Provide the Jacobian for a planar 2R arm at $q = [\pi/4, \pi/6]^T$ with links $l_1 = l_2 = 0.3$ m.

Friction Cone Visualization Exercise:

1. A robot finger exerts a normal force $F_n = 8$ N on a surface with friction coefficient $\mu = 0.6$. Plot the friction cone boundaries in the tangential plane using MATLAB or Python.
2. For the same contact, generate 1000 random valid force vectors within the friction cone and display them as a 3D point cloud.
3. A sensor measures $F = [2.1, -1.3, 7.8]^T$ N. Determine if this lies within the friction cone and calculate the safety margin $\frac{\mu F_n - \|F_t\|}{\mu F_n}$.

5.2.3 Singularity detection: Avoiding the danger zones

[Singularity detection: Avoiding the danger zones]

Jacobian matrix singularity detection Exercise: Given a 3-DOF robotic arm with the following Jacobian matrix:

$$J = \begin{bmatrix} \cos(q_1) + \sin(q_2) & -\sin(q_1) \cos(q_3) & 0 \\ \sin(q_1) \cos(q_2) & \cos(q_1) \sin(q_3) & -1 \\ 0 & \sin(q_2) \cos(q_3) & \cos(q_1) \end{bmatrix}$$

1. Compute the determinant symbolically using MATLAB's Symbolic Math Toolbox

2. Identify all joint configurations $[q_1, q_2, q_3]$ that make the determinant zero
3. For each singular configuration, suggest one practical workaround

Parallel robot singularity analysis Exercise: A 3-RPR parallel manipulator has its platform pose described by:

```
platform_pose = [x; y; phi]
```

1. Derive the condition number of the Jacobian matrix as a function of ϕ
2. Plot the condition number versus ϕ from 0 to 2π radians
3. Identify the ϕ values where the condition number exceeds 1000
4. Propose a path planning strategy to avoid these high-condition-number regions

Singularity-robust inverse kinematics Exercise: Implement a singularity-robust inverse kinematics solver in Python:

```
def sr_inverse_kinematics(J, dx, lambda=0.1):
    # Your implementation here
```

1. Use the damped least-squares method with regularization parameter λ
2. Test with the Jacobian from the first exercise at $q_1=\pi/2$, $q_2=0$, $q_3=\pi/4$
3. Compare the solution with and without damping when near singularity
4. Plot the error norm versus λ for values from 0.01 to 1.0

5.3 Jacobian-Based Control

5.3.1 Resolved-rate control implementation

[Resolved-rate control implementation]

Jacobian matrix computation Exercise: Given a 3-DOF robotic arm with joint angles θ_1, θ_2 , and θ_3 :

1. Derive the analytical Jacobian matrix for the end-effector position $[x, y, z]$.
2. Write a Python function `is_singular(theta_1, theta_2)` that returns `True` if the configuration is singular.
3. Calculate the Jacobian numerically for $\theta_1 = 0.5$, $\theta_2 = 1.2$, $\theta_3 = -0.8$ with link lengths $l_1 = 1.0$, $l_2 = 0.8$, $l_3 = 0.5$.

Singularity avoidance Exercise:

For a 2-DOF planar manipulator with link lengths $l_1 = 1.0$ and $l_2 = 0.6$:

1. Identify all singular configurations by analyzing the determinant of the Jacobian matrix.

2. Write a Python function `is_singular(theta_1, theta_2)` that returns `True` if the configuration is singular.
3. Implement a resolved-rate control scheme that automatically reduces velocity when approaching singularities.

Trajectory tracking Exercise:

Given a desired end-effector trajectory $\mathbf{x}_d(t) = [0.5\cos(t), 0.5\sin(t), 0.1t]$:

1. Derive the required joint velocities using resolved-rate control with a sampling time $\Delta t = 0.01$.
2. Implement a simulation in Python that tracks this trajectory for 10 seconds.
3. Plot the tracking error $\|\mathbf{x}_d(t) - \mathbf{x}(t)\|$ over time and analyze the results.

5.3.2 Null-space control Doing two things at once

[Null-space control: Doing two things at once]

[Joint-space optimization with null-space projection] Exercise:

1. Given a 7-DOF robotic arm with Jacobian matrix \mathbf{J} (size 6×7), compute the null-space projector matrix \mathbf{N} using the pseudoinverse.
2. Implement a Python function that accepts \mathbf{J} and returns the null-space projector \mathbf{N} . Use `numpy.linalg.pinv` for the pseudoinverse.
3. The primary task is end-effector velocity $\mathbf{v}_{\text{desired}} = [0.1, 0, 0, 0, 0, 0]$. The secondary task is joint velocity $\mathbf{q}_{\text{dot}0} = [0, 0.5, 0, 0, 0, 0, 0]$. Compute the combined joint velocities using null-space projection.

[Redundant manipulator obstacle avoidance] Exercise:

1. A 6-DOF manipulator has a critical joint at $\mathbf{q}_{\text{crit}} = [0, 0.8, 0, 0, 0, 0]$. Design a cost function $H(\mathbf{q})$ that penalizes proximity to this configuration.
2. Derive the gradient $dH/d\mathbf{q}$ for your cost function.
3. Given the Jacobian \mathbf{J} and desired end-effector velocity $\mathbf{v}_{\text{desired}}$, implement the null-space projection to avoid \mathbf{q}_{crit} while tracking $\mathbf{v}_{\text{desired}}$.

[Dual-task mobile robot control] Exercise:

1. A differential drive robot has primary task Jacobian \mathbf{J}_p for trajectory tracking and secondary task Jacobian \mathbf{J}_s for keeping the heading at 45 degrees.
2. Compute the combined control law using null-space projection, ensuring the heading task doesn't interfere with trajectory tracking.
3. Implement this in Python with $\mathbf{J}_p = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ and $\mathbf{J}_s = \begin{bmatrix} 0 & 1 \end{bmatrix}$. Use $\mathbf{v}_p = [0.5, 0.5]$ and $\mathbf{v}_s = [0.3]$.

5.3.3 When to use pseudoinverses vs optimization

[When to use pseudoinverses vs optimization]

Linear system solving with pseudoinverse Exercise:

1. Given the matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ and vector $\mathbf{b} = \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}$, compute the pseudoinverse A^+ using singular value decomposition.
2. Verify that A^+ satisfies the Moore-Penrose conditions.
3. Use A^+ to find the least-squares solution \mathbf{x} to $A\mathbf{x} = \mathbf{b}$.
4. Compare the result with solving $\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2^2$ using gradient descent (implement in Python with `numpy`).

Optimization for ill-conditioned systems Exercise:

1. Implement Tikhonov regularization to solve $\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2^2 + \lambda \|\mathbf{x}\|_2^2$, where $A = \begin{bmatrix} 1 & 1 \\ 1 & 1.0001 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 2 \\ 2.0001 \end{bmatrix}$, and $\lambda = 0.1$.
2. Plot the solution norm $\|\mathbf{x}\|_2$ versus λ for $\lambda \in [10^{-6}, 10^0]$ (logarithmic scale).
3. Explain why optimization is preferred over pseudoinverse for this problem.

Robust regression comparison Exercise:

1. Generate a dataset with outliers: 50 points from $y = 2x + 1 + \mathcal{N}(0, 0.5)$ and 5 outliers at $x = 0$ with $y = 10$.
2. Solve using (a) pseudoinverse and (b) Huber loss optimization $\min_{\mathbf{x}} \sum_i \phi(y_i - \mathbf{a}_i^T \mathbf{x})$, where ϕ is the Huber function with $\delta = 1.0$.
3. Plot both regression lines and the data points using `matplotlib`.

Chapter 6

Mobile Robot Kinematics: Robots That Drive

6.1 Wheeled Mobile Robots

6.1.1 Differential drive: The bread and butter

[Differential drive: The bread and butter]

[Kinematics of Differential Drive] Exercise: A differential drive robot has wheel radius $r = 0.05$ m and wheelbase $L = 0.20$ m. The robot moves forward with left wheel angular velocity $\omega_l = 4$ rad/s and right wheel angular velocity $\omega_r = 6$ rad/s.

1. Calculate the linear velocities of the left and right wheels.
2. Compute the robot's translational velocity v and angular velocity ω .
3. Determine the instantaneous radius of curvature R of the robot's path.

[Odometry Calculation] Exercise: A differential drive robot starts at pose $(x, y, \theta) = (0, 0, 0)$. Over $\Delta t = 2$ s, the wheel encoders report the following displacements: Left wheel $\Delta s_l = 0.3$ m, Right wheel $\Delta s_r = 0.5$ m.

1. Calculate the average translational displacement Δs and rotational displacement $\Delta \theta$.
2. Update the robot's pose using the odometry equations.
3. Compute the final position (x', y') and orientation θ' of the robot.

[Motor Control Implementation] Exercise: Write a Python function to control a differential drive robot using PWM signals. The robot has a wheelbase $L = 0.18$ m and max PWM duty cycle 255.

```
def set_robot_velocity(v, omega):  
    # v: translational velocity (m/s)  
    # omega: angular velocity (rad/s)  
    # Returns: (left_pwm, right_pwm) tuple  
    pass
```

1. Derive the wheel velocity equations for v and ω .

2. Implement the function to convert v and ω to left/right PWM values.
3. Handle edge cases where PWM values exceed the maximum duty cycle.

6.1.2 Skid-steer kinematics: When you have tracks

[Skid-steer kinematics: When you have tracks]

[Track slip calculation] Exercise: A skid-steer robot with track width $w = 0.5$ m and track length $L = 0.8$ m moves forward at $v = 0.3$ m/s while experiencing 15% slip on the left track and 10% slip on the right track.

1. Calculate the effective velocities v_{left} and v_{right} accounting for slip.
2. Derive the instantaneous turning radius R using the slip-adjusted velocities.
3. Compute the resulting angular velocity ω of the robot.

[Track tension analysis] Exercise: A skid-steer robot with mass $m = 150$ kg accelerates at $a = 0.4$ m/s² on a concrete surface (coefficient of friction $\mu = 0.6$).

1. Calculate the minimum required track tension T_{min} to prevent slippage during acceleration.
2. Determine the power P needed per track motor if the robot moves at $v = 1.2$ m/s during acceleration.
3. Assuming 85% motor efficiency, find the electrical power P_{elec} drawn from the battery.

[Track wear simulation] Exercise: A skid-steer robot's track wear is modeled by the function $\text{wear_rate} = k * F_N * v_{\text{rel}}$ where $k = 0.002$ (wear coefficient), F_N is normal force, and v_{rel} is relative track-ground speed.

1. Write a Python function `calculate_wear()` that computes wear rate given F_N and v_{rel} .
2. Simulate wear over 1000 cycles where F_N varies between 200-500 N and v_{rel} between 0.1-0.5 m/s.
3. Plot the cumulative wear using `matplotlib` with properly labeled axes.

```
# Sample starter code
import numpy as np
import matplotlib.pyplot as plt
def calculate_wear(F_N, v_rel):
    return 0.002 * F_N * v_rel
```

6.1.3 Using ROS navigation stack: Standing on giants' shoulders

[Using ROS navigation stack: Standing on giants' shoulders]

[Costmap Configuration] Exercise: Modify the global costmap parameters for a differential drive robot to optimize navigation in a cluttered environment.

- Open the `global_costmap_params.yaml` file in your ROS package.
- Set `inflation_radius` to 0.5 meters and `cost_scaling_factor` to 10.0.
- Change the `obstacle_range` parameter to 3.0 meters to limit sensor input range.
- Add a new layer for static map with `track_unknown_space` set to `true`.
- Test the changes by running `roslaunch your_package navigation.launch`.

[AMCL Localization Tuning] Exercise: Improve AMCL localization accuracy for a robot in a dynamic office environment.

- Locate the `amcl_params.yaml` file in your ROS package.
- Adjust `laser_min_range` to 0.1 and `laser_max_range` to 8.0 meters.
- Set `kld_err` to 0.01 and `kld_z` to 0.99 for particle filter optimization.
- Increase `max_particles` to 5000 for better localization in large spaces.
- Verify improvements by observing the particle cloud in RViz during navigation.

[Navigation Stack Integration] Exercise: Implement a custom recovery behavior when the robot gets stuck.

- Create a new ROS node called `recovery_behavior.cpp`.
- Subscribe to the `/move_base/status` topic to detect failure states.
- When stuck, publish a twist command to rotate 45 degrees clockwise.
- After rotation, attempt to move forward 0.3 meters before resuming navigation.
- Test using `rostopic` with intentionally blocked paths in your test environment.

6.2 Advanced Mobile Platforms

6.2.1 Omniwheels and mecanum: Moving sideways

[Omniwheels and mecanum: Moving sideways]

[Mecanum Wheel Kinematics] Exercise: A robot with four mecanum wheels has the following wheel velocities: $v_1 = 0.5$ m/s, $v_2 = -0.3$ m/s, $v_3 = 0.4$ m/s, $v_4 = -0.2$ m/s. The wheel diameter is 0.1 m and the wheel angle is 45 degrees.

1. Calculate the robot's translational velocity in the x-direction.
2. Calculate the robot's translational velocity in the y-direction.

3. Determine the robot's rotational velocity (ω).

[Omniwheel Control Implementation] Exercise: A three-omniwheel robot uses the following inverse kinematics matrix:

```
[ 0.866  -0.5    1;
 -0.866  -0.5    1;
  0       1     1 ]
```

1. Write Python code to compute wheel speeds given desired $v_x = 0.2$, $v_y = -0.1$, and $\omega = 0.5$.
2. Validate your solution by checking if the output wheel speeds produce the desired motion.
3. Modify the code to handle wheel speed saturation at ± 1.0 m/s.

[Mecanum Robot Path Planning] Exercise: A warehouse robot with mecanum wheels must move 3 m east, then 2 m north, then rotate 90 degrees clockwise.

1. Calculate the required wheel speeds for each segment of motion.
2. Determine the total time required if maximum wheel speed is 1.2 m/s.
3. Plot the robot's path using any software, showing wheel velocity vectors during turns.

6.2.2 KDL for mobile base control

[KDL for mobile base control]

[Kinematic Modeling] Exercise:

1. Derive the forward kinematics equations for a differential-drive mobile base with wheel radius r and wheel separation L .
2. Implement the forward kinematics in Python using `numpy`:

```
def forward_kinematics(phi_l, phi_r, r, L, dt):
    # Your code here
```

3. Calculate the robot's pose after 2 seconds given $r=0.1\text{m}$, $L=0.5\text{m}$, $\dot{\phi}_l=2\text{rad/s}$, and $\dot{\phi}_r=1\text{rad/s}$.

[Trajectory Tracking] Exercise:

1. Design a proportional controller for a unicycle model to track a straight-line trajectory along the x-axis.
2. Implement the controller in ROS using `geometry_msgs/Twist` messages:

```
def control_callback(current_pose, desired_pose):
    # Your code here
```

3. Tune the controller gains to achieve less than 5cm steady-state error for a 2m straight-line path.

[Obstacle Avoidance] Exercise:

1. Modify the differential-drive kinematic model to include a safety margin d_{\min} around obstacles.
2. Implement a potential field method in C++ using `Eigen` vectors:


```
Eigen::Vector2d compute_repulsive_force(Eigen::Vector2d obstacle_pos,
                                         double d_min);
```
3. Test your implementation with an obstacle at (1.0, 0.5) and $d_{\min}=0.3\text{m}$, plotting the resulting force vectors.

6.2.3 Simulation with MuJoCo: Testing before building

[Simulation with MuJoCo: Testing before building]

[Inverse Dynamics Validation] Exercise:

1. Implement a simple 2D pendulum model in MuJoCo with a mass of 1 kg and length of 1 m.
2. Use `mj_inverse` to compute the required torque at the joint when the pendulum is at 45 degrees from vertical.
3. Compare the computed torque with the theoretical value calculated using $\tau = m \cdot g \cdot l \cdot \sin(\theta)$.
4. Vary the angle from 0 to 90 degrees in 10-degree increments and plot the error between MuJoCo and theoretical values.

[Contact Force Analysis] Exercise:

1. Create a MuJoCo scene with a box (1x1x1 m) falling onto a fixed platform from 2 m height.
2. Enable contact logging using `mjvOption.flags[7] = 1` before simulation.
3. Run the simulation for 2 seconds and extract the normal force during impact.
4. Calculate the theoretical impulse using conservation of momentum and compare with MuJoCo's results.
5. Repeat with a coefficient of restitution of 0.5 and analyze the force profile changes.

[Actuator Saturation Test] Exercise:

1. Model a robotic arm with two revolute joints in MuJoCo, each with a maximum torque of 10 Nm.
2. Implement a PD controller ($k_p=100$, $k_d=10$) to track a 90-degree step input.
3. Record the actual torque output and joint angles during the transient response.
4. Identify the time instances when actuator saturation occurs.
5. Modify the controller gains to eliminate saturation while maintaining rise time under 0.5 seconds.

6.3 Mobile Manipulation

6.3.1 Combining mobile base and manipulator kinematics

[Combining mobile base and manipulator kinematics]

[Forward kinematics of a mobile manipulator] Exercise: A differential drive mobile robot has a 2-DOF planar manipulator mounted on top. The mobile base has wheel radius $r = 0.1$ m and wheel separation $L = 0.5$ m. The manipulator has link lengths $l_1 = 0.3$ m and $l_2 = 0.2$ m.

1. Calculate the mobile base's transformation matrix after moving 2 m forward with both wheels rotating at 10 rad/s.
2. Compute the manipulator's end-effector position relative to its base when joint angles are $\theta_1 = 45^\circ$ and $\theta_2 = 30^\circ$.
3. Combine both transformations to find the end-effector's world coordinates, assuming the manipulator base is located 0.4 m forward from the mobile base's center.

[Jacobian for mobile manipulator velocity] Exercise: A holonomic mobile base with Mecanum wheels carries a 3-DOF RRR manipulator. The base velocity is given by $\dot{x}_b = 0.2$ m/s, $\dot{y}_b = -0.1$ m/s, $\dot{\theta}_b = 0.05$ rad/s. The manipulator has link lengths $l = [0.4, 0.3, 0.2]$ m and current joint angles $q = [30^\circ, 45^\circ, 60^\circ]$.

1. Compute the geometric Jacobian matrix for the manipulator's end-effector.
2. Calculate the end-effector's velocity contribution from the mobile base motion.
3. Combine both velocity components to find the total end-effector twist in world coordinates.

[Obstacle avoidance with kinematic constraints] Exercise: A mobile manipulator must move its end-effector to $p_{goal} = [1.5, 0.8, 0.6]$ m while avoiding a cylindrical obstacle at $[1.0, 0.5]$ m with radius 0.3 m.

1. Formulate the inequality constraint to keep the mobile base (radius 0.4 m) outside the obstacle's safety margin.
2. Write the kinematic constraint ensuring the manipulator's second link doesn't collide with the obstacle (treat link as line segment).
3. Implement a Python function that checks both constraints given the system state $(x_b, y_b, \theta_b, q_{arm})$:

```
def check_constraints(x_b, y_b, theta_b, q_arm):
    # Your implementation here
```

6.3.2 Coordinated motion planning

[Coordinated motion planning]

[Trajectory generation for two robots] Exercise: A robotic arm and an autonomous mobile robot must move simultaneously to avoid collision. The arm's end-effector follows a path defined by:

$$x(t) = 0.5 \cos \pi(2t/T)$$

$$y(t) = 0.3 \sin \pi(2t/T)$$

The mobile robot moves along the x-axis with constant velocity v_{robot} .

1. Calculate the minimum safe distance between robots when $T = 10$ s and $v_{\text{robot}} = 0.2$ m/s
2. Modify the arm's trajectory to maintain at least 0.5 m clearance at all times
3. Implement the collision check in Python using `numpy.linspace` for time discretization

[Multi-drone formation control] Exercise: Three quadrotors must maintain an equilateral triangle formation with 2 m sides while moving.

1. Derive the required velocity vectors when the formation center moves at $[1.5, 0.0]$ m/s
2. Design a PID controller for relative position maintenance using `leader-follower` approach
3. Simulate the system for 10 seconds with 0.1 s timesteps in MATLAB

[Industrial robot coordination] Exercise: Two 6-DOF industrial robots must transfer parts between conveyors without interference.

1. Calculate the workspace overlap volume given joint limits $[-90^\circ, 90^\circ]$ for all joints
2. Generate waypoints for both robots using cubic splines with 0.5 s intervals
3. Verify timing synchronization using ROS `moveit` trajectory messages

6.3.3 Real-world constraints: Cables, battery life, and physics

[Real-world constraints: Cables, battery life, and physics]

[Cable resistance and voltage drop] Exercise: A sensor node is powered by a 5V supply over a 20m cable with resistance $0.1\Omega/\text{m}$. The node draws 500mA continuously.

1. Calculate the total cable resistance.
2. Determine the voltage drop across the cable.
3. Find the actual voltage reaching the sensor node.
4. If the node requires at least 4.5V to operate, suggest two solutions to address the voltage drop.

[Battery lifetime estimation] Exercise: A wireless IoT device uses a 3.7V 2000mAh Li-ion battery and has the following power characteristics:

- Active mode: 120mA for 5ms every 10s
- Sleep mode: 1.5 μ A between active periods

1. Calculate the average current consumption.
2. Estimate the battery lifetime in days.
3. The device firmware is updated to reduce active time to 3ms. Recalculate the new lifetime.
4. Identify which power state dominates the battery drain and why.

[Thermal constraints in enclosures] Exercise: An embedded system in a sealed enclosure has:

- Total power dissipation: 6.8W
- Enclosure surface area: 0.25m²
- Maximum allowed surface temperature: 60°C
- Ambient temperature: 25°C

1. Calculate the required thermal resistance of the enclosure.
2. The actual enclosure has $R_{th} = 4.5^{\circ}\text{C/W}$. Compute the expected surface temperature.
3. Determine if forced cooling is needed and justify your answer.
4. List two methods to improve thermal performance without changing the enclosure size.

Chapter 7

Simulation Workflows: Test Before You Wreck

7.1 Simulation-First Development

7.1.1 PyBullet joint control: Fast iteration cycles

[PyBullet joint control: Fast iteration cycles]

[Joint Position Control] Exercise: Implement a PID controller for a single revolute joint in PyBullet.

1. Create a PyBullet simulation with a simple 1-DOF robotic arm (single link and joint).
2. Set the joint position target to 1.57 radians (90 degrees).
3. Implement a PID controller using `p.setJointMotorControl2()` with `controlMode=p.POSITION_CONTROL`.
4. Tune the PID gains (K_p , K_i , K_d) to achieve settling time under 2 seconds with less than 5% overshoot.
5. Plot the joint angle vs time using `matplotlib` to verify performance.

[Torque-based Control] Exercise: Compare direct torque control vs position control for joint stabilization.

1. Create a PyBullet scene with a pendulum (single revolute joint with gravity).
2. Implement two control methods: position control using `p.POSITION_CONTROL` and direct torque control using `p.TORQUE_CONTROL`.
3. For torque control, compute required torque using $\tau = K_p \cdot (\text{target_position} - \text{current_position})$.
4. Measure and compare the energy consumption (integral of squared torque) for both methods.
5. Record the maximum deviation from target position during a 10-second simulation.

[Iterative Tuning] Exercise: Develop a systematic tuning procedure for joint controllers.

1. Create a test rig with a 1-DOF system in PyBullet, adding random noise to joint position feedback.
2. Write a Python function that automatically runs simulations with different PID parameters.
3. Implement a performance metric combining settling time, overshoot, and steady-state error.
4. Use a grid search to find the best PID parameters for three different payload masses (0.1kg, 1kg, 10kg).
5. Export the results as a CSV file showing the optimal gains for each mass.

7.1.2 MuJoCo kinematics APIs: When accuracy matters

[MuJoCo kinematics APIs: When accuracy matters]

[Forward Kinematics Validation] Exercise: Implement a MuJoCo forward kinematics verification test for a 3-DOF robotic arm.

1. Load the MuJoCo model `arm_3dof.xml` using `mjv_makeModel()`.
2. Set joint angles to $q = [0.5, -0.3, 1.2]$ radians via `mj_data->qpos`.
3. Compute end-effector position using `mj_forward()` followed by `mj_getSitePosition()`.
4. Manually calculate the expected position using DH parameters from the model.
5. Compare results with absolute tolerance $1e-6$ and print "PASS" or "FAIL".

[Jacobian Numerical Verification] Exercise: Validate MuJoCo's analytic Jacobian against finite differences.

1. Create a 2-DOF planar manipulator model in MuJoCo.
2. Use `mj_jacSite()` to get the analytic Jacobian at $q = [0.1, 0.4]$.
3. Implement central differences with $\delta = 1e-7$ to compute numerical Jacobian.
4. For each Jacobian element, verify `abs(analytic - numeric) < 1e-5`.
5. Report the maximum observed error across all elements.

[Singularity Detection] Exercise: Detect kinematic singularities using MuJoCo's Jacobian functions.

1. Load the MuJoCo model `ur5.xml`.
2. Sample 100 random joint configurations using `mj_data->qpos`.
3. For each configuration, compute the Jacobian via `mj_jacSite()`.

4. Calculate the matrix condition number using SVD (`mj_svd()`).
5. Flag configurations where condition number exceeds `1e4` as singular.
6. Visualize singular configurations using `mjv_updateScene()`.

7.1.3 Gazebo integration: The ROS ecosystem approach

[Gazebo integration: The ROS ecosystem approach]

[Gazebo-ROS Bridge Setup] Exercise:

1. Install the `gazebo_ros_pkgs` package using `apt` on Ubuntu.
2. Launch an empty Gazebo world with ROS integration using `roslaunch gazebo_ros empty_world.launch`.
3. Verify the ROS-Gazebo bridge is active by checking the list of published topics with `rostopic list`.
4. Spawn a simple URDF model (e.g., a box) using the `gazebo_ros spawn` service.
5. Add a plugin to the URDF model that subscribes to a ROS topic and applies forces to the model.

[ROS Control in Gazebo] Exercise:

1. Create a URDF robot with two revolute joints and corresponding `transmission` tags.
2. Configure the `gazebo_ros_control` plugin in the URDF file.
3. Launch the robot in Gazebo with the `ros_control` controllers.
4. Write a ROS node that publishes joint position commands to move the robot.
5. Visualize the joint states in RViz while the robot is moving in Gazebo.

[Sensor Simulation and Data Processing] Exercise:

1. Add a simulated IMU sensor to a robot URDF using the `gazebo_ros_imu_sensor` plugin.
2. Launch the robot in Gazebo and verify IMU data is published to a ROS topic.
3. Write a Python node that subscribes to the IMU data and calculates roll, pitch, and yaw angles.
4. Add a noise model to the IMU sensor in the URDF file using Gazebo's noise parameters.
5. Plot the raw and filtered IMU data using `rqt_plot`.

7.2 Debugging and Visualization

7.2.1 Plotting trajectories: Making motion visible

[Plotting trajectories: Making motion visible]

Projectile Motion Visualization Exercise:

1. A projectile is launched with initial velocity $v_0 = 25 \text{ m/s}$ at angle $\theta = 45 \text{ deg}$.
2. Write Python code using `matplotlib` to plot the trajectory (y vs x).
3. Assume gravitational acceleration $g = 9.81 \text{ m/s}^2$ and no air resistance.
4. Calculate and label the maximum height and range on your plot.
5. Use `numpy.linspace` to generate time values with at least 100 points.

Pendulum Phase Space Exercise:

1. A simple pendulum has length $L = 1.0 \text{ m}$ and initial angle $\theta_0 = 0.2 \text{ rad}$.
2. Using the small-angle approximation, write MATLAB code to plot the phase portrait (angular velocity vs angle).
3. Set your time axis to cover exactly 5 periods of oscillation.
4. Include grid lines and label both axes with appropriate units.
5. Use `ode45` with relative tolerance $1e-6$ for numerical integration.

Multi-body Trajectory Comparison Exercise:

1. Three particles start at the origin with velocities $v_1 = [2, 1]$, $v_2 = [1, 3]$, $v_3 = [4, 0.5] \text{ m/s}$.
2. Create a single plot showing all three trajectories under constant acceleration $a = [0, -0.5] \text{ m/s}^2$.
3. Use different line styles (solid, dashed, dotted) for each trajectory.
4. Add a legend showing which line style corresponds to which initial velocity.
5. Set equal scaling for both axes to preserve geometric relationships.

7.2.2 Real-time visualization tools

[Real-time visualization tools]

WebSocket Data Display Exercise: Create a Python script using Flask-SocketIO that:

1. Sets up a basic web server with route `/`
2. Broadcasts random temperature values between 20°C and 30°C every 2 seconds
3. Displays values in a browser using Chart.js with time-series formatting

4. Includes client-side JavaScript to handle disconnection events

ROS Topic Visualization Exercise: Using ROS and RViz:

1. Create a publisher node that generates dummy lidar scan data (use `sensor_msgs/LaserScan`)
2. Configure RViz to display the scan with intensity-based coloring
3. Add a dynamic reconfigure parameter to change scan resolution
4. Record a bag file with 10 seconds of scan data

Embedded System Dashboard Exercise: For an STM32 microcontroller:

1. Implement UART transmission of accelerometer data (format: "X:val,Y:val,Z:val\n")
2. Create a Processing sketch to plot all three axes in real-time
3. Add threshold indicators that turn red when any axis exceeds $\pm 2g$
4. Implement serial port selection via dropdown menu

7.2.3 Comparing simulation vs hardware: Reality checks

[Comparing simulation vs hardware: Reality checks]

ADC Resolution Verification Exercise:

1. Connect a 12-bit ADC to a microcontroller using SPI communication.
2. Generate a known DC voltage between 0-3.3V using a precision power supply.
3. Write firmware to sample this voltage 100 times using `adc_read()`.
4. Calculate the mean and standard deviation of the readings.
5. Compare the observed resolution with the theoretical LSB size (3.3V/4096).
6. Explain any discrepancy greater than 2 LSBs between theory and measurement.

Digital Filter Implementation Check Exercise:

1. Implement a 4th-order Butterworth low-pass filter in Python with 100Hz cutoff.
2. Apply this filter to a synthetic signal containing 50Hz and 150Hz components.
3. Implement the same filter on a DSP using fixed-point arithmetic (Q15 format).
4. Feed identical input data to both implementations.
5. Compare the output waveforms and RMS error.
6. Measure the DSP's actual execution time versus the sample period.

Wireless RSSI Calibration Exercise:

1. Set up two identical 2.4GHz transceivers 1 meter apart in an anechoic chamber.
2. Record the RSSI values from 1000 packets at 1dBm transmit power.
3. Repeat measurements at 5dBm, 10dBm, and 15dBm transmit powers.
4. Plot measured RSSI versus theoretical free-space path loss.
5. Calculate the error between expected and measured RSSI values.
6. Propose calibration factors for the RSSI reporting function.

7.3 Simulation to Reality Transfer

7.3.1 What works in simulation but fails on hardware

[What works in simulation but fails on hardware]

[PWM Signal Jitter] Exercise: A microcontroller generates a 10 kHz PWM signal in simulation with perfect timing. On hardware, the signal exhibits jitter of ± 200 ns.

1. Calculate the percentage deviation from the ideal 50% duty cycle caused by this jitter.
2. Modify the Arduino code below to implement hardware-based PWM on Timer1 instead of `analogWrite()`.

```
void setup() {
    // Original software PWM
    pinMode(9, OUTPUT);
}

void loop() {
    analogWrite(9, 128); // 50% duty cycle
}
```

3. Propose two hardware solutions to reduce jitter when driving a MOSFET gate.

[Sensor Noise Floor] Exercise: An ADC reads 12-bit values from a temperature sensor in simulation. Hardware measurements show a noise floor of 8 LSB RMS.

1. Convert the noise floor to millivolts given a 3.3V reference voltage.
2. Rewrite the Python code below to apply a moving average filter with window size 5.

```
def read_sensor():
    return adc.read_channel(0)
```

3. Specify the minimum temperature resolution in $^{\circ}\text{C}$ if the sensor has a 100°C range.

[Motor Control Latency] Exercise: A PID controller for a DC motor works perfectly in simulation. On hardware, the control loop misses deadlines due to 2ms interrupt latency.

1. Calculate the maximum allowable control frequency to stay within 5% latency budget.
2. Identify which line in the ISR below causes the longest delay and rewrite it.

```
void TIM2_IRQHandler() {
    TIM2->SR &= ~TIM_SR_UIF;
    current = ADC1->DR;
    error = setpoint - current;
    integral += error;
    PWM3->CCR1 = Kp*error + Ki*integral;
}
```

3. Replace the floating-point operations with fixed-point arithmetic using Q15 format.

7.3.2 Tuning simulation parameters for reality

[Tuning simulation parameters for reality]

[Adjusting spring-mass-damper parameters] Exercise:

1. Implement a spring-mass-damper system in MATLAB using `ode45` with default parameters: $m = 1$, $k = 10$, $c = 0.5$
2. Modify the damping coefficient c to achieve critical damping
3. Plot displacement vs time for underdamped, critically damped, and overdamped cases on the same axes
4. Determine the settling time for each case using `find` and logical indexing

[Optimizing PID controller gains] Exercise:

1. Design a PID controller for the transfer function $G(s) = 1/(s^2 + 2s + 1)$
2. Use Ziegler-Nichols tuning rules to determine initial K_p , K_i , and K_d values
3. Implement the controller in Simulink with a unit step input
4. Adjust the gains to reduce overshoot below 5% while maintaining rise time under 2 seconds
5. Export the step response data and calculate ISE (Integral Squared Error)

[Calibrating sensor noise models] Exercise:

1. Generate synthetic accelerometer data with true sinusoidal motion at 5 Hz
2. Add Gaussian white noise with $\sigma = 0.1 \text{ m/s}^2$ using `randn`
3. Implement a moving average filter with window sizes 5, 10, and 20 samples
4. Calculate RMS error between filtered and true signals for each window size
5. Determine the optimal window size that minimizes both RMS error and phase lag

7.3.3 Using simulation data to train real controllers

[Using simulation data to train real controllers]

[PID Tuning with Simulated Data] Exercise:

1. Simulate a DC motor in MATLAB using `tf([1], [0.1 1])` as the plant model.
2. Collect step response data for 5 different PID gain sets in a matrix `sim_data`.
3. Design a neural network with 3 inputs (`K_p`, `K_i`, `K_d`) and 2 outputs (rise time, settling time).
4. Train the network using `fitnet` with your `sim_data`, reserving 20% for validation.
5. Export the trained network to ONNX format using `exportONNXNetwork`.

[ROS Controller Deployment] Exercise:

1. Create a Gazebo simulation of a differential drive robot with lidar sensors.
2. Record 1 hour of simulated sensor data and control inputs using `rosv bag record`.
3. Preprocess the bag files into CSV format using `ros2 bag export`.
4. Train a PyTorch CNN to predict steering angles from lidar scans.
5. Implement the trained model in a ROS node using `torchscript`.

[Hardware-in-the-Loop Validation] Exercise:

1. Connect a physical servo motor to a Simulink Real-Time target machine.
2. Configure a hardware-in-the-loop testbench using `xPC Target`.
3. Stream simulated position commands from MATLAB to the physical servo.
4. Record actual vs commanded positions at 1kHz sampling rate.
5. Calculate and plot the RMS error between simulated and real responses.

Chapter 8

Performance and Real-Time Concerns

8.1 From Prototype to Production

8.1.1 When to port Python prototypes to C++

[When to port Python prototypes to C++]

Performance Bottleneck Analysis Exercise: Exercise:

1. Write a Python function `compute_fibonacci(n)` that calculates the n th Fibonacci number using recursion.
2. Time its execution for $n=35$ using Python's `timeit` module.
3. Implement the same recursive algorithm in C++ and measure its runtime for $n=35$.
4. Compare the execution times and calculate the speedup ratio (`Python_time/C++_time`).
5. Modify the C++ version to use memoization and repeat the timing measurements.

Real-time Constraint Evaluation Exercise: Exercise:

1. Create a Python script that simulates processing 1000 sensor readings with `numpy.random.normal`.
2. Measure the 99th percentile latency using `time.perf_counter`.
3. Port the processing to C++ using the `library`'s normal distribution.
4. Compare the latency distributions between implementations.
5. Determine the maximum sampling rate where Python still meets a 1ms latency target.

Memory-bound Operation Conversion Exercise: Exercise:

1. Implement a Python matrix multiplication using nested lists for 1000x1000 matrices.
2. Profile memory usage with `memory_profiler` during operation.
3. Rewrite the operation in C++ using contiguous memory allocation with `new[]`.
4. Measure peak memory consumption in both implementations.
5. Calculate the memory savings percentage when using the C++ version.

8.1.2 Profiling performance: Finding the real bottlenecks

[Profiling performance: Finding the real bottlenecks]

CPU Cache Profiling Exercise: Exercise:

1. Write a C program that allocates two 1D integer arrays, `data_a` and `data_b`, each of size 1,000,000.
2. Initialize both arrays with random values between 0 and 255.
3. Measure and compare the execution time of summing all elements in `data_a` versus summing every 16th element in `data_b`.
4. Use `clock_gettime()` with `CLOCK_MONOTONIC` for timing measurements.
5. Explain the performance difference using CPU cache concepts.

Memory Access Pattern Exercise: Exercise:

1. Implement a Python function `matrix_multiply` that multiplies two 1024x1024 matrices using nested loops.
2. Profile the function using `cProfile` and identify the top 3 most time-consuming operations.
3. Rewrite the function to transpose one matrix before multiplication and measure the speedup.
4. Include the `timeit` results for both implementations with 10 repetitions.
5. State whether the improvement aligns with expectations from cache locality principles.

Disk I/O Bottleneck Exercise: Exercise:

1. Create a Java program that reads a 1GB binary file sequentially versus randomly using `FileInputStream`.
2. Use `System.nanoTime()` to measure read throughput in MB/s for both approaches.
3. Repeat the experiment with a 4KB buffer size and a 1MB buffer size.
4. Plot the results using `gnuplot` or `matplotlib` showing throughput vs. access pattern.
5. Propose an optimal buffer size based on typical HDD/SSD block sizes.

8.1.3 Memory management in real-time systems

[Memory management in real-time systems]

[Static allocation trade-offs] Exercise: A real-time system must manage sensor data from 10 identical temperature sensors. Each sensor requires a 32-byte buffer to store its latest reading. Compare static and dynamic allocation for this scenario by:

- Calculating the total memory required if using static allocation.

- Listing two advantages of static allocation for this specific case.
- Identifying one potential drawback if the number of sensors changes frequently.

[Stack analysis] Exercise: Analyze the stack usage of the following real-time task:

```
void RT_task() {
    int32_t sensor_values[8];
    float calibration_factors[4];
    struct { uint16_t id; uint8_t status; } device;
    // ... (no recursive calls)
}
```

- Calculate the minimum stack size required assuming 32-bit architecture.
- Explain why this calculation is critical for real-time systems.
- Suggest one method to verify the actual stack usage during testing.

[Memory pool implementation] Exercise: Design a fixed-size memory pool for a real-time system that needs to allocate:

- 20 blocks of 64 bytes each for network packets
- 15 blocks of 32 bytes each for control messages

```
// Provide the C struct definitions for:
// 1. The memory pool control structure
// 2. The block header structure
// Assume 32-bit alignment requirements
```

8.2 Real-Time Kinematics

8.2.1 Determinism in kinematic computations

[Determinism in kinematic computations]

[Position analysis of a 4-bar linkage] Exercise:

1. Given a 4-bar linkage with link lengths $L_1 = 10$ cm, $L_2 = 4$ cm, $L_3 = 8$ cm, and $L_4 = 7$ cm, compute the possible output angles θ_3 when the input angle $\theta_2 = 45^\circ$.
2. Implement the position analysis solution in Python using `numpy`. Your code should output both possible configurations (open and crossed).
3. Verify your results by checking the loop closure equation $L_2 \exp(i\theta_2) + L_3 \exp(i\theta_3) - L_4 \exp(i\theta_4) - L_1 = 0$.

[Velocity analysis of a slider-crank mechanism] Exercise:

1. For a slider-crank mechanism with crank length $r = 3$ cm, connecting rod length $l = 10$ cm, and crank angular velocity $\omega_2 = 5$ rad/s, calculate the slider velocity when $\theta_2 = 30^\circ$.

2. Derive the velocity equation using the vector loop method and show all steps.
3. Write a MATLAB script that plots slider velocity versus crank angle for one full revolution.

[Acceleration in a quick-return mechanism] Exercise:

1. A Whitworth quick-return mechanism has $L_{OA} = 5$ cm, $L_{AB} = 15$ cm, and $L_{OQ} = 10$ cm. Compute the angular acceleration of link AB when $\theta_2 = 60^\circ$ and $\alpha_2 = 2$ rad/s².
2. Draw the acceleration polygon to scale and label all vectors.
3. Create a Simulink model that outputs the angular acceleration of link AB for one cycle.

8.2.2 Library usage in hard real-time control loops

[Library usage in hard real-time control loops]

[Timer Interrupt Service Routine] Exercise: Design a timer interrupt service routine (ISR) using the `TimerOne` library for Arduino to toggle an LED at 1 kHz. The ISR must measure and store the execution time of the main control loop in microseconds.

1. Include the `TimerOne.h` library and declare a volatile global variable `loop_time_us`.
2. Set up `Timer1` to trigger an interrupt every 1 ms (1 kHz frequency).
3. Implement the ISR to toggle pin 13 and read `micros()` before and after the main `loop()`.
4. Calculate the execution time as the difference between microsecond readings.
5. Verify the timing using an oscilloscope on pin 13.

[RTOS Task Synchronization] Exercise: Create two FreeRTOS tasks that share a thread-safe queue for real-time motor control commands. Use the `QueueHandle_t` API to enforce hard timing constraints.

1. Declare a queue with capacity for 10 `uint16_t` values using `xQueueCreate()`.
2. Implement `Task1` to generate PWM duty cycle values (0-100) every 5 ms.
3. Implement `Task2` to consume values from the queue and write to `analogWrite()` within 1 ms of receipt.
4. Use `uxTaskGetStackHighWaterMark()` to verify stack space for both tasks.
5. Measure the worst-case latency using `xTaskGetTickCount()`.

[DMA-accelerated Sensor Reading] Exercise: Configure the STM32 HAL library to read an ADC sensor via DMA while maintaining a 100 μ s control loop.

1. Declare a `DMA_HandleTypeDef` and `ADC_HandleTypeDef` for the STM32F4.
2. Use `CubeMX` to generate initialization code for ADC1 on channel 5 with DMA in circular mode.

3. Implement a 100 μ s delay using `HAL_Delay()` with SysTick configured at 1 MHz.
4. Verify the ADC readings don't stall the main loop by toggling a GPIO pin.
5. Plot the timing jitter using a logic analyzer on the GPIO pin.

8.2.3 Practical trade-offs: Accuracy vs speed vs reliability

[Practical trade-offs: Accuracy vs speed vs reliability]

Signal Processing Filter Implementation Exercise:

1. Design a low-pass FIR filter with cutoff frequency 0.2 times the sampling rate using the window method
2. Implement the filter in Python using `scipy.signal.firwin` with 31 taps
3. Measure execution time using `timeit` for filtering a 10-second audio signal sampled at 44.1 kHz
4. Reduce the tap count to 15 and compare both the frequency response and execution time
5. Explain which implementation you would choose for a real-time application and why

Database Index Optimization Exercise:

1. Create a PostgreSQL table `customer_transactions` with fields: `transaction_id`, `customer_id`, `amount`, and `timestamp`
2. Insert 1 million random records using `generate_series` and random data generation
3. Execute a query finding all transactions for a specific `customer_id` between two dates without any indexes
4. Create appropriate indexes and measure the query execution time difference
5. Discuss the trade-off between query speed and insert performance when adding indexes

Neural Network Pruning Exercise:

1. Train a simple CNN on MNIST using PyTorch with at least 3 convolutional layers
2. Measure baseline accuracy and inference time on 1000 test samples
3. Apply magnitude-based pruning to remove 20% of the smallest weights
4. Retrain the pruned model while keeping zeroed weights frozen
5. Compare the accuracy, model size, and inference time before and after pruning

8.3 Optimization Strategies

8.3.1 Caching and precomputation techniques

[Caching and precomputation techniques]

LRU Cache Implementation Exercise: Implement a Least Recently Used (LRU) cache in Python with the following specifications:

- The cache must support `get(key)` and `put(key, value)` operations.
- Both operations must run in $O(1)$ average time complexity.
- The cache has a fixed capacity specified during initialization.
- When the cache reaches capacity, `put` must evict the least recently used item.
- Include a `get_cache_state()` method that returns all items in order of recent use.

Use a combination of a doubly linked list and hash map. Provide test cases for:

- Cache eviction when exceeding capacity.
- Updating existing keys without eviction.
- Retrieving non-existent keys.

Matrix Chain Multiplication Exercise: Given a sequence of matrix dimensions, precompute the optimal parenthesization for multiplication:

- Implement the dynamic programming solution to find the minimum scalar multiplications.
- Store intermediate results in a memoization table to avoid recomputation.
- Print the optimal parenthesization as a string (e.g., `"((A(BC))D)"`).
- Handle edge cases: single matrix, incompatible dimensions, and empty input.

Test with these matrix chains:

$10 \times 30, 30 \times 5, 5 \times 60$

$10 \times 3, 3 \times 12, 12 \times 5$

100×100

Prime Number Sieve Exercise: Implement the Sieve of Eratosthenes with caching optimizations:

- Precompute all primes up to a given limit using the sieve algorithm.
- Store the results in a class-level cache to avoid recomputation.
- Add a method `is_prime(n)` that checks the cache first.

- If n exceeds the precomputed limit, extend the sieve dynamically.
- Benchmark the cached vs. non-cached versions for $n=1,000,000$.

Include error handling for:

- Negative inputs.
- Non-integer inputs.
- Memory limits when caching large ranges.

8.3.2 Parallel processing for multi-robot systems

[Parallel processing for multi-robot systems]

[Deadlock avoidance] Exercise: A multi-robot warehouse system uses parallel processing to coordinate 4 robots (`robot_1` to `robot_4`). Each robot requires exclusive access to two resources: a charging station (`charger_A` or `charger_B`) and a loading dock (`dock_X` or `dock_Y`). The current resource allocation is:

- `robot_1` holds `charger_A` and requests `dock_X`
- `robot_2` holds `dock_X` and requests `charger_B`
- `robot_3` holds `charger_B` and requests `dock_Y`
- `robot_4` holds `dock_Y` and requests `charger_A`

Draw a resource allocation graph and determine if the system is in deadlock. If deadlock exists, propose two solutions.

[Task partitioning] Exercise: A swarm of 8 cleaning robots must process 1200 map grid cells in parallel. Each robot has identical processing capability. The workload consists of:

- 400 obstacle cells requiring 3ms each
- 600 floor cells requiring 1ms each
- 200 special surface cells requiring 5ms each

Calculate the optimal task partitioning to minimize total processing time. Provide the cell distribution per robot and the expected makespan.

[Message passing implementation] Exercise: Implement a Python function using multiprocessing to coordinate two robots via message passing:

```
def robot_comm(pipe_end, robot_id):
    # Robot 1 sends its battery level (85%) and requests status
    # Robot 2 responds with its battery level (72%) and "ACK"
    # Both robots print received messages
    # Use pipe_end.send() and pipe_end.recv()
```

Create the parent process that initializes two `Pipe` connections and starts both robot processes.

8.3.3 Hardware acceleration: GPUs and specialized processors

[Hardware acceleration: GPUs and specialized processors]

GPU Matrix Multiplication Exercise: Implement a matrix multiplication kernel using CUDA or OpenCL. Compare its performance against a CPU implementation.

1. Write a CPU function in C/C++ that multiplies two 1024x1024 float matrices.
2. Implement a GPU kernel using `__global__` functions in CUDA or `__kernel` in OpenCL.
3. Use shared memory to optimize tile-based matrix multiplication.
4. Measure execution time for both implementations using 10 repeated trials.
5. Calculate the speedup factor (GPU time / CPU time) and report memory transfer overhead.

TPU Quantization Exercise: Simulate tensor processing unit (TPU) behavior by implementing 8-bit quantization.

1. Create a Python function that converts FP32 weights to INT8 using linear quantization.
2. Implement matrix-vector multiplication using the quantized weights.
3. Dequantize the results and calculate the mean squared error versus FP32.
4. Profile memory savings by comparing `.nbytes` of original and quantized NumPy arrays.
5. Modify the quantization to use symmetric vs asymmetric schemes and compare errors.

AI Accelerator Profiling Exercise: Analyze performance characteristics of a neural network on different hardware.

1. Install TensorFlow with GPU support and run `tf.config.list_physical_devices()`.
2. Train a small CNN on MNIST using CPU, GPU, and TPU (if available) backends.
3. Use `tf.profiler` to measure FLOPs, memory usage, and execution time.
4. Create bar charts comparing the metrics across hardware platforms.
5. Identify which layer operations show the largest speedup on accelerators.

Chapter 9

When Things Break (And They Will)

9.1 Singularity Hell

9.1.1 Detecting singularities with API tools

[Detecting singularities with API tools]

[Numerical stability in matrix inversion] Exercise:

1. Compute the condition number of matrix $A = \begin{bmatrix} 1 & 2 \\ 2 & 4.0001 \end{bmatrix}$ using the 2-norm.
2. Write Python code using `numpy.linalg` to verify your result from step 1.
3. Solve $Ax = b$ where $b = \begin{bmatrix} 3 \\ 6.0001 \end{bmatrix}$ and observe the sensitivity to small perturbations.

[API-based singularity detection] Exercise:

1. Install the `scipy.linalg` package and use `det()` to check for singularity in $B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$.
2. Write a function that returns "near-singular" if the condition number exceeds 10^8 .
3. Test your function on $C = \begin{bmatrix} 1 & 0 \\ 0 & 1e-9 \end{bmatrix}$ and report the output.

[Practical rank deficiency analysis] Exercise:

1. Load the dataset "singular_data.csv" (assumed to contain a 100x100 matrix) using `pandas.read_csv`.
2. Compute the matrix rank with `numpy.linalg.matrix_rank` at tolerance `tol=1e-5`.
3. Plot the singular values using `matplotlib.pyplot.semilogy` and identify the drop-off point.

9.1.2 Avoidance strategies that actually work

[Avoidance strategies that actually work]

Buffer overflow prevention Exercise: Write a C function that safely copies a string using `strncpy` with proper null-termination.

1. Declare a destination buffer of size 32 bytes
2. Implement boundary checks before copying
3. Force null-termination of the destination buffer
4. Handle the case where source exceeds destination size
5. Return 0 on success or -1 on failure

```
int safe_copy(char* dest, const char* src, size_t dest_size);
```

SQL injection mitigation Exercise: Create a Python function that executes parameterized SQL queries safely.

1. Use `sqlite3` module's parameter substitution
2. Validate input types before query execution
3. Implement error handling for database operations
4. Return query results as a list of dictionaries
5. Close database connections properly

```
def safe_query(db_path: str, query: str, params: tuple) -> list:
```