Application of Reinforcement Learning Algorithms for Motion planning on Hexapod and Quadruped Robots

Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T/BITS F422T/BITS F423T/BITS F424T Thesis

By
Tejas Mathur Sriganesh
ID No. : 2021A4PS1415P

Under the supervision of:

Dr. Abdalla Swikir
Senior Scientist, MIRMI, Technische Universität München

&

Dr. Amit Rajnarayan Singh
Assistant Professor, Dept of Mechanical Engineering, BITS Pilani

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
(October, 2024)

Acknowledgements

I would like to express my sincere gratitude to those who have supported and guided me throughout the development of this mid-semester thesis, titled **"Application of Reinforcement Learning Algorithms for Motion Planning on Hexapod and Quadruped Robots."**

First and foremost, I am deeply grateful to **Dr. Abdalla Swikir** for providing me with the opportunity to undertake this project and for his encouragement throughout. I also extend my sincere thanks to **Dr. Amit Rajnarayan Singh** for his invaluable support and guidance during the course of this research. His insights have been instrumental in shaping the project and keeping it on the right track.

A special thank you goes to **Dr. Quang Hoan Le**, my mentor at the Technical University of Munich, whose guidance and expertise were crucial to the progress and success of this project. The few months I spent in Munich were a truly unique and enriching experience, both academically and personally. My time there allowed me to immerse myself in a highly stimulating academic environment, which laid the groundwork for the research to come and helped me grow as a researcher. The experience of living and working in Munich has been invaluable in refining my skills and understanding, significantly contributing to my work on this thesis.

Finally, I would like to extend my heartfelt gratitude to my **family and friends** for their unwavering support and encouragement throughout this period. Their belief in me has been a constant source of motivation, and I am truly grateful for their patience and understanding.

To all who have contributed to this endeavor, I offer my deepest thanks. Your encouragement and support have been indispensable, and I look forward to continuing this research with the same enthusiasm and dedication.

Abstract

This report presents the progress of ongoing research focused on the application of reinforcement learning algorithms for motion planning on hexapod and quadruped robots. The primary objective is to develop effective locomotion strategies by integrating reinforcement learning techniques within simulated environments and transitioning them to physical models.

During the initial phase, simulations were conducted using a pre-designed hexapod robot in PyBullet, achieving successful motion planning and movement. Building on this foundation, the research progressed to the design and manufacture of a custom quadruped robot. With the quadruped model now completed, future work will involve implementing a similar reinforcement learning approach to enable the newly created robot to achieve stable and adaptive locomotion.

The expected outcomes of this project include a fully functional quadruped robot capable of dynamic movement, validated through both simulation and real-world testing, and insights into the adaptability of reinforcement learning algorithms across different robotic platforms.

Table of Contents

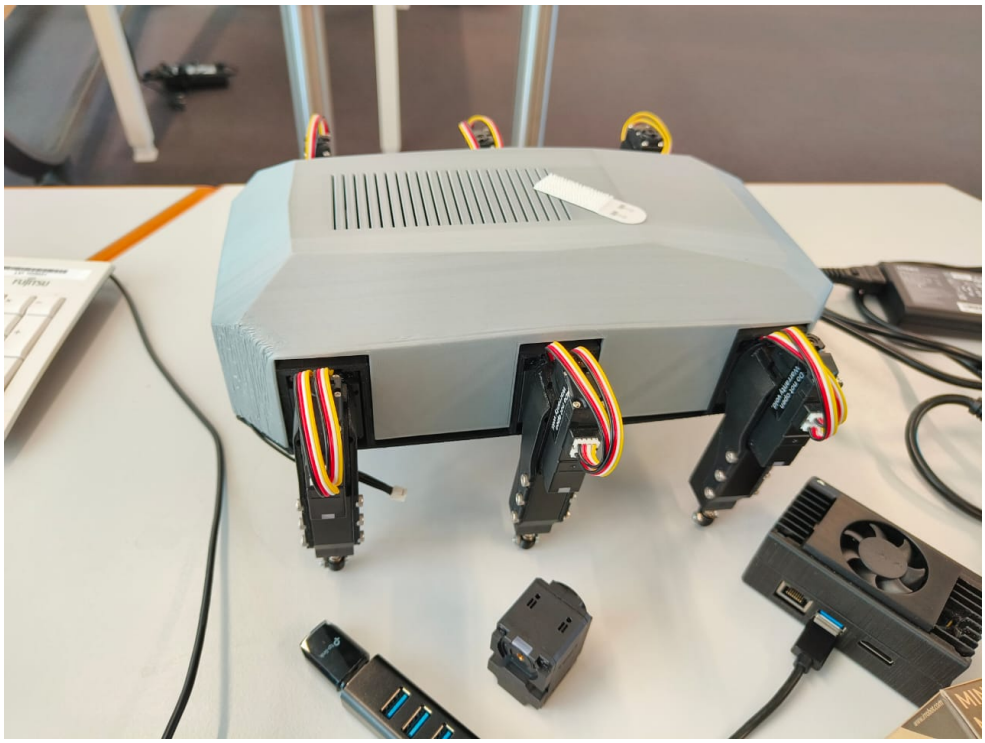
Introduction

Robotic locomotion and navigation have witnessed remarkable advancements with the rise of sophisticated algorithms and intelligent control systems. Among these advancements, **Reinforcement Learning (RL)** has emerged as a powerful tool for developing adaptive motion strategies in complex environments. Reinforcement learning algorithms allow robots to learn and adapt autonomously by interacting with their surroundings, making them particularly suitable for dynamic and uncertain scenarios.

In the realm of legged robots, such as **hexapods** and **quadrupeds**, effective motion planning is critical for achieving stable and agile locomotion. These robots must navigate complex terrains, balance efficiently, and respond quickly to environmental changes—challenges that traditional control systems often struggle to address. Reinforcement learning offers a promising solution by enabling robots to optimize their movement through iterative trial and error, leading to the discovery of optimal motion patterns.

Mobile robots, particularly those with multi-legged configurations, have a wide range of practical applications. In **search and rescue missions**, legged robots can manoeuvre through rubble and uneven terrain, reaching areas inaccessible to wheeled robots or human rescuers. In the **agricultural sector**, they can be deployed for precision farming, performing tasks like soil monitoring, planting, and harvesting in diverse environments. **Military and surveillance operations** also benefit from such robots, as they can be used for reconnaissance in hazardous areas without risking human life. Additionally, in **planetary exploration**, hexapod and quadruped robots provide robust solutions for traversing extraterrestrial landscapes, where uneven surfaces and unpredictable conditions are common.

The goal of this project is to leverage reinforcement learning techniques for enhancing the locomotion capabilities of both hexapod and quadruped robots. The initial phase focused on the simulation of a pre-designed



hexapod robot in **PyBullet**, where basic movement was successfully achieved. Building upon this progress, the project shifted towards the design and manufacture of a custom quadruped robot, setting the stage for

Fig 1: hexapod robot at the lab

future experiments that will apply similar learning strategies to enable the quadruped to walk and navigate effectively.

This research not only aims to contribute to the field of legged robotics but also seeks to explore the adaptability and scalability of reinforcement learning across different robotic platforms. The findings are expected to provide valuable insights into the feasibility of using RL for motion planning in robots with varying configurations and degrees of freedom, paving the way for more capable and versatile mobile robots in a variety of challenging environments.

Hexapod Design

The first week I joined Dr Hoan wanted me to learn how to control the motors used on the Hexapod in the lab.

The hexapod we were using was a robot that was using 6 mighty zap linear actuators and 6 dynamixel xl430-w250-t servos.

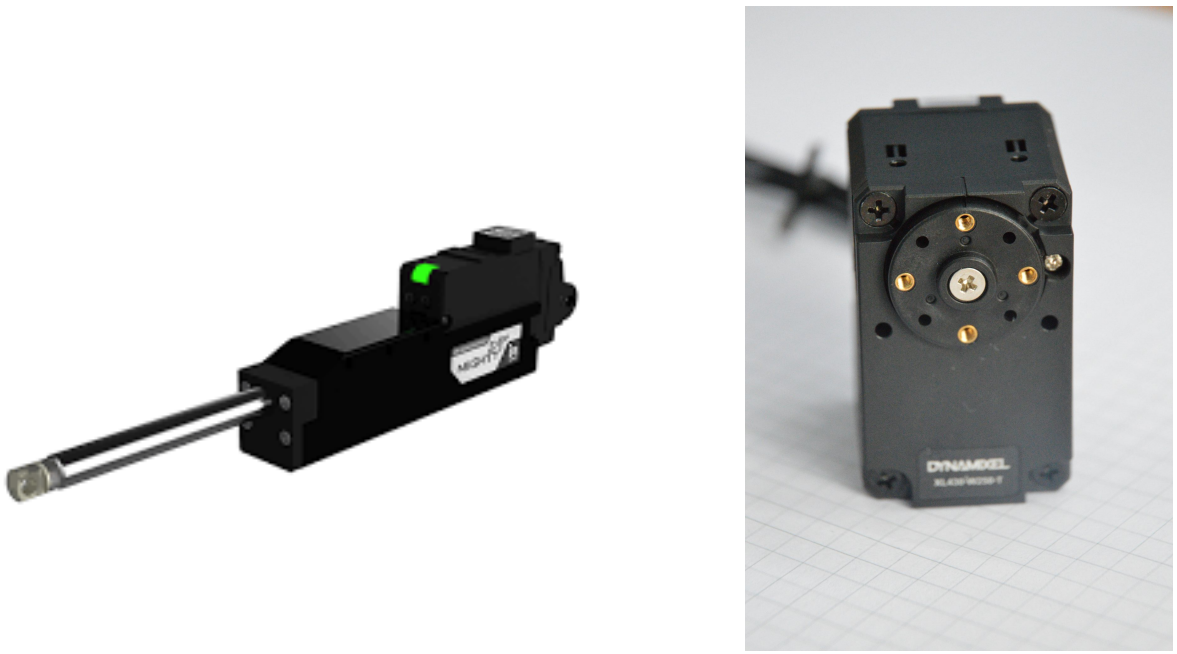


Figure2:liner and rotary Actuators

The robot also contains a Jetson nano as an on board computer that we had to take remote and control later on.

Figure 3: Jetson nano

The whole robot has 3d printed parts which puts the production cost of this bot at around 600 euros for the whole bot without the Jetson included, which is much lower in cost compared to robots available on the market as of now.

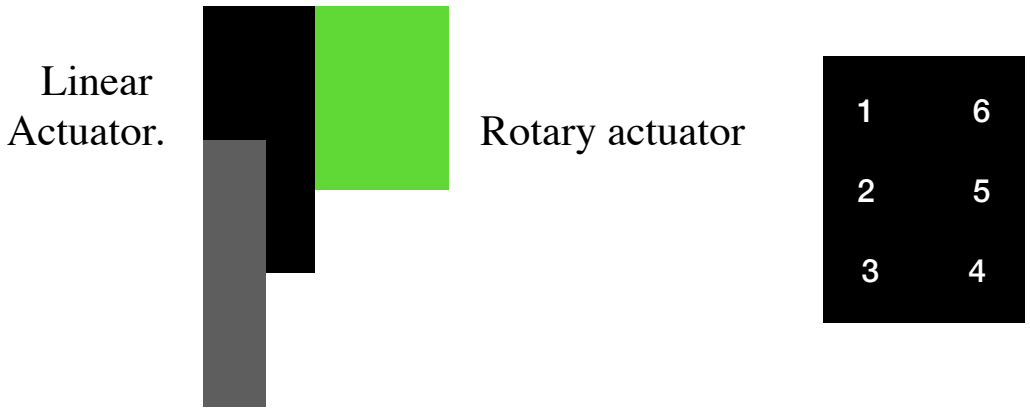


Figure 4: Leg structure and top view leg positions on bot

Each leg is a 3 DoF RRP arm which gives our bot a crab like look. Both motors communicate using an RS 485 3 pin protocol
Which we convert into a usb using a u2d2 board and an irusb02 module

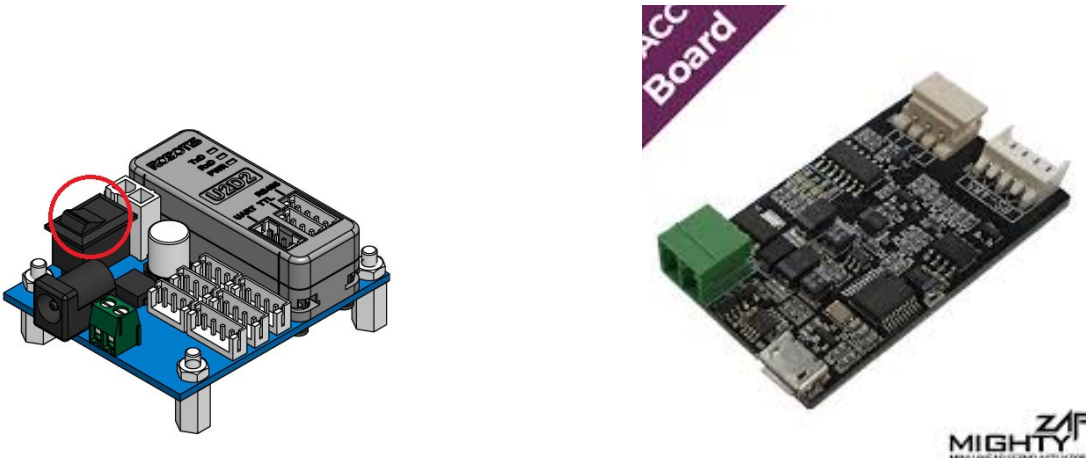


Figure 5:u2d2 module and irusb02 converter module

A 20000mAh power bank is used to power both the robots. We need power for both usb converters and the Jetson as well.

Motor Control

The first week I was there I was tasked with figuring out how to control the motors. I first figured it out for the dynamixel servo.

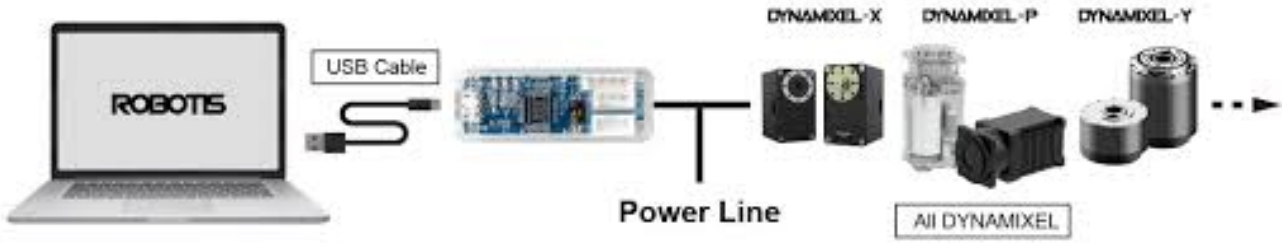


Figure 6: control layout for dynamixel servo

The layout is very simple we have the motor that gives an rs485 protocol o/p that we connect to the u2d2. This takes power from an external source and has a micro usb slot to connect it to the on board pc.

After we connect it to the pc we can control the rotary servo one of 2 ways either using the dynamixel wizard or using the python api. The wizard is a software by the manufacturers of the servo. It lets us edit variables of the servo such as its id to address it, baud rate to communicate with it and multiple other functions such as max velocity goal position etc.

W do the same thing using the python api the main difference is we can use conditionals in the second option which make it more suitable.

The motor communicates with our pc using an array type data object where each address on the byte of data has a specific function, for example we can change address 64 between 0/1 tenable and disable torque or change address 116 using data for setting goal position from 0 to 4095.

Below link is a video of the dynamixel being controlled:

https://drive.google.com/file/d/1aYYupq4hop55oo-uaGfE6MZjFsJJwFdS/view?usp=sharing

Next I moved onto the mighty zap linear actuator.
it had a similar layout but the motor was swapped and we use an irusb02 instead of the u2d2 to convert the rs-485 protocol to usb and to provide power to motor. But this motor only has a python api.

It has a much easier to use api as well compared to dynamixel.

Example code for mighty zap control:

```
importPythonLibMightyZap
importtime
MightyZap = PythonLibMightyZap
Actuator_ID = 0
MightyZap.OpenMightyZap('COM3',57600)
for i in range(0,2):
 MightyZap.goalPosition(Actuator_ID,4095)
 time.sleep(3)
 print(MightyZap.presentPosition(0))
 MightyZap.goalPosition(Actuator_ID,0)
 time.sleep(3)
 print(MightyZap.presentPosition(0))
```

This code extends the actuator fully and then retracts it twice

Indoor GPS design

The **Marvelmind Indoor GPS system** is an advanced positioning technology designed to provide precise location tracking in indoor environments where traditional GPS signals are unreliable. It consists of a network of fixed base stations (anchors) that communicate with mobile tags, allowing for accurate real-time positioning within a defined area.

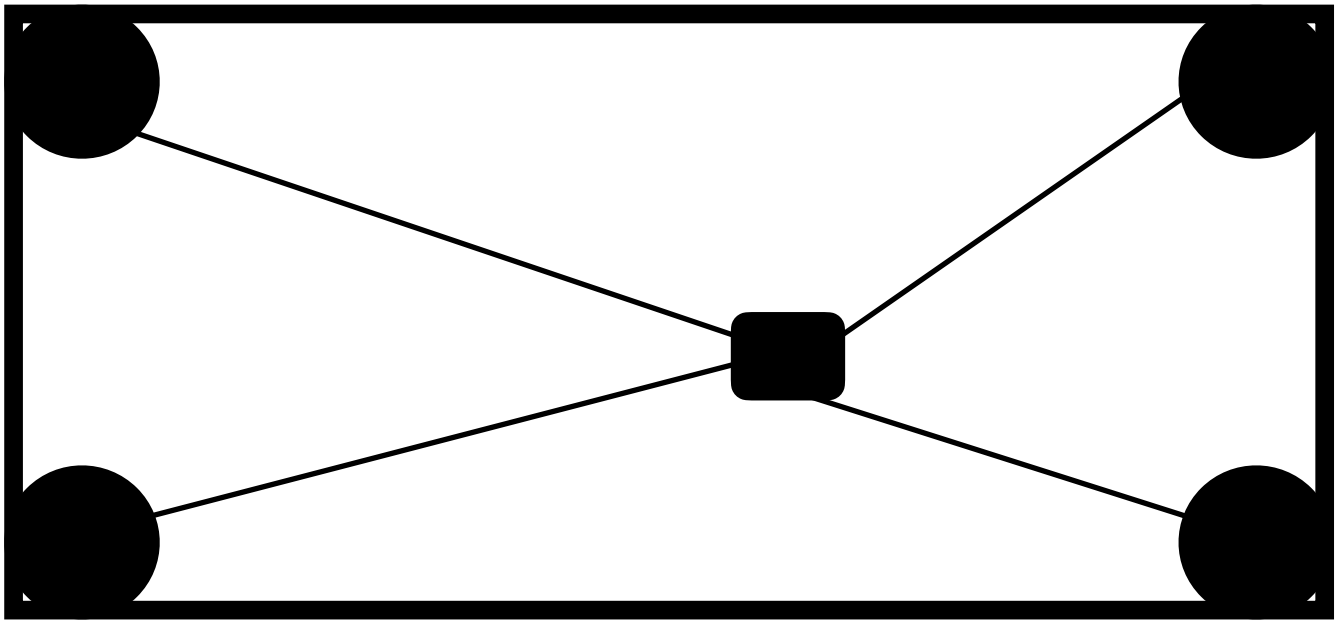


Figure 7: moving beacon positioning using stationary beacons

These beacons help provide x,y and z coordinates of the moving beacon which help us chart the path of the hexapod while we make it walk.
these beacons also have inbuilt IMU sensors and accelerometers which help us estimate velocity and orientation of our bot. This helps detect when our bot is fallen as an added benefit which helps us later on.

The setup of this system was relatively easy I pasted the beacons two 4 corners of our work space and added the moving beacon on the robot and parallel connected the antenna module to a battery.
Once I applied the python api provided by marvel mind I was getting a constant data stream of position and orientation data.

This data was very messy though and had lots of erroneous points so added a Kalman filter to filter out odd data points which helped me get more accurate data points.

Basically the code eliminates less probable points from the set giving us a cleaner path.

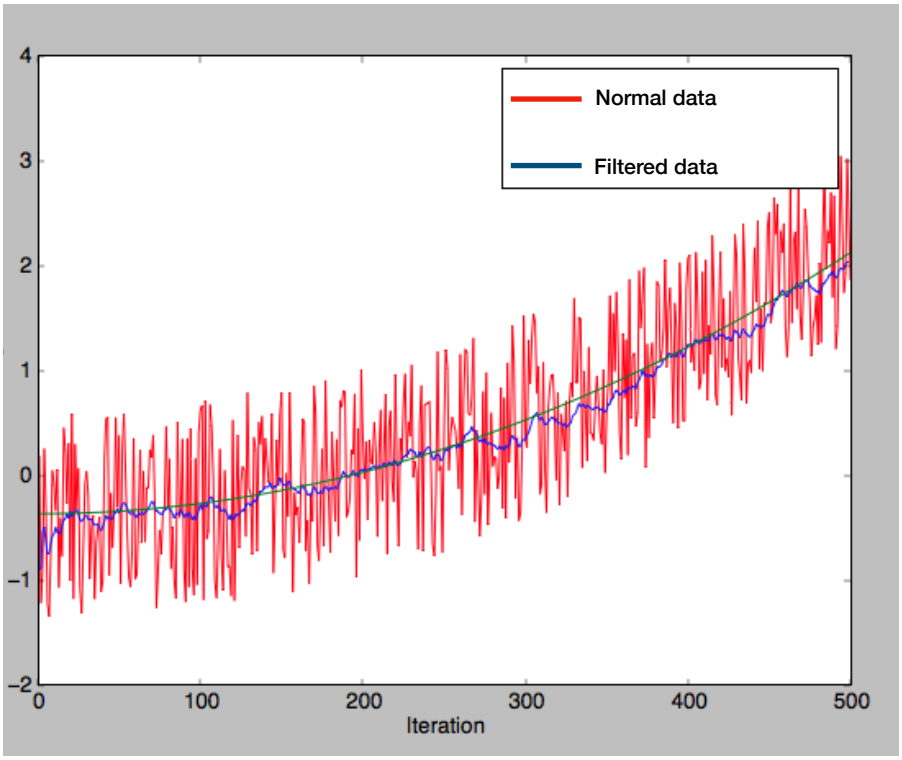


Figure 8: noisy and filter data

As we see in the graph we eliminate the improbable points from the set only leaving out probable points for us.

Quadruped Design

I was asked to design a quadruped robot based on the design of the microspot bot in order to be able to add dynamixel servos as they have feedback unlike the original servos of that design so I worked on this parallel for some time while implementing stuff on the hexapod. The final robot had a longer body than the original giving it a sausage like appearance. So we named this new robot the dachshund based on the German breed of dog that has the same build.

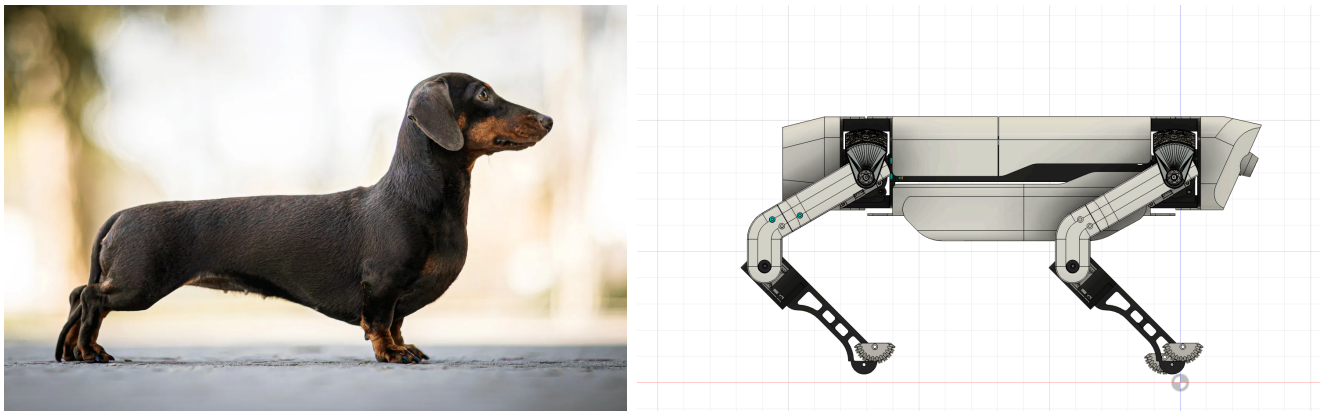


Figure 10: The Dachshund

The new design came with wider servo mounts to incorporate the motors same as the hexapod, below there are photos of the leg itself from 2 angles to see the design better.

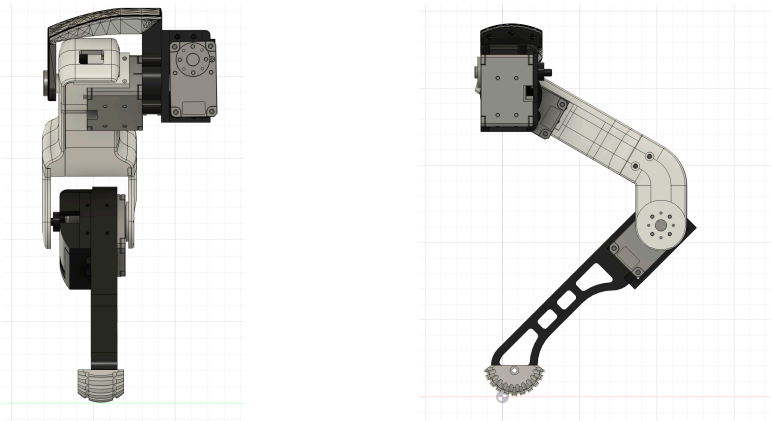


Figure 11: leg of the dachshund

The inside of the robot has to contain the Jetson,the usb boards and the power source as in the power bank we are using.

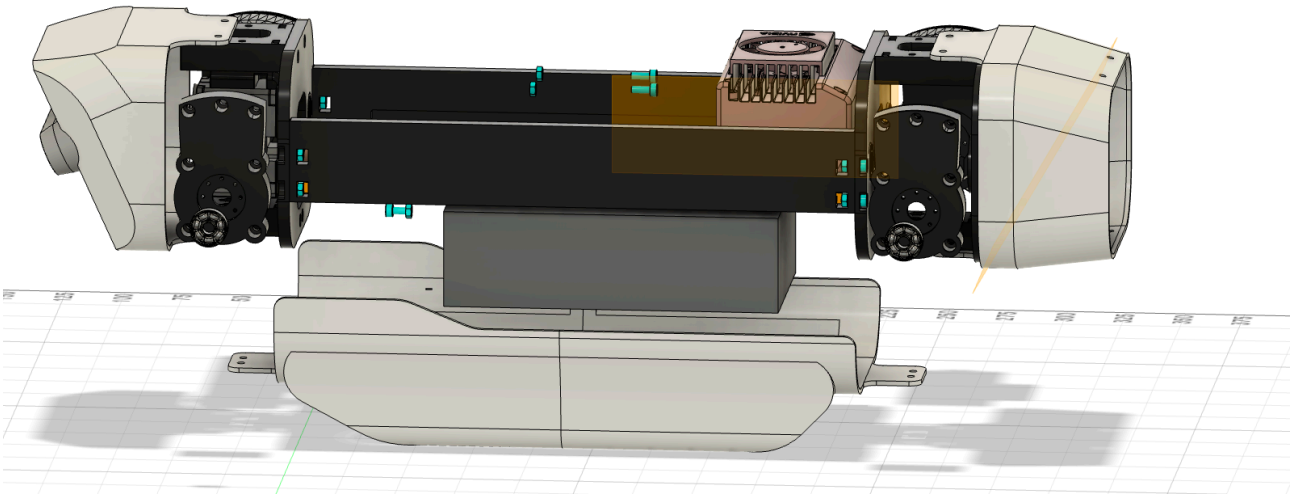


Figure 12: torso of the dachshund

This torso is specifically elongated to encapsulate all our needed parts.we also added a belly like depression to the bot as a functional location to store the battery. We got the idea when we saw pictures of dachshunds with bloated stomachs online while researching for the name as this would help centralise the weight of the battery which was our heaviest item also in our robot also this leaves more space above the rack for the battery management system and a usb extender for the Jetson so we could plug in more slots into the bot.

Quadruped Fabricaion and Assembly

All of the Dachshund's parts have been additively manufactured with PLA using 3d printers.all the parts have been kept at an infill of 30% and we used reality ender pro and prusa mks+ printers.
We took each individual part in subgroups and slices using prusa and Cura slicer softwares. Once all the parts were fabricated we assembled them. Almost all fastenings on the dachshund are M3 or M2.5 screws and nuts with hex heads.

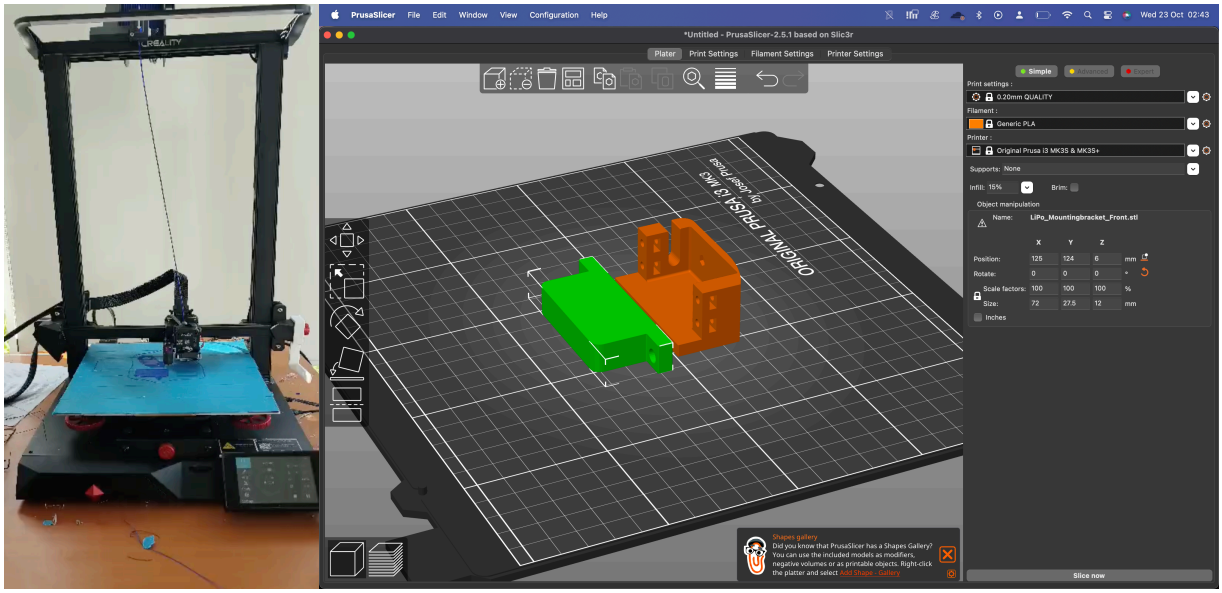


Figure13: 3d Printer and Prusa Slicer

Pros and Cons of Using RL

Using **Reinforcement Learning (RL)** for motion planning offers several advantages over traditional manual motion planning approaches. Here are the key benefits:

1. Adaptability to Complex and Dynamic Environments

- **RL Advantage**: Reinforcement learning enables robots to adapt their behavior based on real-time feedback from the environment. This is particularly useful for navigating unpredictable terrains or dynamic settings where pre-defined rules may not apply.
- **Manual Motion Planning Limitation**: Traditional approaches rely on predefined algorithms or trajectories, which are less flexible and often require manual adjustments when conditions change.

2. Automatic Discovery of Optimal Solutions

- **RL Advantage**: Through trial and error, RL can autonomously discover optimal or near-optimal motion strategies. It can identify solutions that might not be intuitive to human designers, potentially outperforming hand-crafted plans.
- **Manual Motion Planning Limitation**: Manual planning depends heavily on human expertise and can result in suboptimal solutions, especially for high-dimensional or highly dynamic problems.

3. Scalability to High-Dimensional Systems

- **RL Advantage**: RL excels in managing robots with many degrees of freedom, such as hexapods or quadrupeds, where the action space is vast and interdependent. It learns to coordinate movements efficiently, which is challenging to achieve manually.
- **Manual Motion Planning Limitation**: The complexity of defining and tuning trajectories increases exponentially with the number of joints and actuators, making manual approaches impractical for complex systems.

4. Real-Time Adaptation

- **RL Advantage**: RL-trained policies can adapt in real-time to changes in the environment or robot dynamics, such as a leg malfunction or slippery surfaces.
- **Manual Motion Planning Limitation**: Static motion plans lack the ability to adjust dynamically, requiring significant reprogramming for new conditions.

5. Reduced Dependency on Accurate Models

- **RL Advantage**: Model-free RL algorithms, such as Q-Learning or Deep Reinforcement Learning, learn directly from interaction data and do not require an accurate mathematical model of the robot or environment.
- **Manual Motion Planning Limitation**: Manual methods often depend on precise kinematic and dynamic models, which are time-consuming to develop and may not accurately reflect real-world conditions.

6. Efficient Use of Computational Resources in Execution

- **RL Advantage**: Once trained, RL policies are computationally efficient during execution, as they rely on simple lookups or neural network inference to decide actions.
- **Manual Motion Planning Limitation**: Manual planning often involves real-time computation of trajectories or path optimization, which can be resource-intensive.

7. Learning from Experience

- **RL Advantage**: RL allows robots to improve their performance over time by learning from past experiences. This capability is especially valuable for long-term deployments where conditions and tasks evolve.
- **Manual Motion Planning Limitation**: Manual planning does not inherently involve any learning; improvements must be introduced explicitly by reprogramming.

8. Handling Stochasticity and Uncertainty

- **RL Advantage**: RL can handle stochastic environments where outcomes are uncertain or probabilistic, as it learns policies that maximize expected rewards.

- **Manual Motion Planning Limitation**: Traditional methods struggle in uncertain environments, requiring additional layers of complexity like probabilistic models or robust control.

While **Reinforcement Learning (RL)** has many advantages, it also comes with several limitations and challenges that can make it less suitable for certain applications or introduce additional complexity. Here are the key cons of using RL:

1. High Computational Requirements

- **Con**: Training RL models requires significant computational resources and time, especially for complex tasks. Simulating thousands of iterations or running trials on hardware can be resource-intensive.
- **Impact**: This can limit accessibility and delay project timelines for teams without access to high-performance computing.

2. Data Inefficiency

- **Con**: RL algorithms often require vast amounts of interaction data to learn effectively. In real-world scenarios, this can result in high costs, increased wear on hardware, or extended training periods.
- **Impact**: Gathering sufficient data for training is impractical or unsafe in many cases, such as when physical robots are involved.

3. Reward Design Challenges

- **Con**: Designing effective reward functions is complex and critical to the success of RL. Poorly designed rewards can lead to unintended or suboptimal behaviors.
- **Impact**: Misaligned rewards can cause robots to prioritize irrelevant actions or fail to achieve the desired task efficiently.

4. Sim-to-Real Transfer Limitations

- **Con**: Policies trained in simulation often fail to work perfectly in real-world environments due to differences in dynamics, noise, and environmental factors, creating a "reality gap."

- **Impact**: Additional tuning and adaptation are often required, increasing development effort and complexity.

5. Safety Concerns During Training

- **Con**: In physical robots, RL training involves exploration, which can lead to unsafe or damaging behaviors (e.g., falling or collisions).
- **Impact**: Such risks can result in hardware damage or require costly interventions during training.

In conclusion, while reinforcement learning offers significant advantages, such as adaptability, optimal motion strategies, and scalability to complex systems, it also comes with notable challenges. High computational demands, inefficiencies in data usage, and risks during training can complicate its implementation. Additionally, the reliance on well-designed reward functions and the challenges of transferring simulation-trained policies to real-world applications underscore the need for careful planning.

Despite these limitations, the flexibility and potential of RL to achieve tasks that are infeasible with traditional methods make it a valuable tool in robotics. By addressing its drawbacks—through robust simulation environments, hybrid approaches, and fine-tuning—RL can be effectively harnessed to unlock new possibilities in motion planning and robotic locomotion.

Implementation Methodology

1. Create the Robot's URDF Model

The **Unified Robot Description Format (URDF)** is used to define the robot's physical and kinematic properties. This includes:

- **Structure**: Defining the body, legs, and joints.
- **Joints**: Specifying the type of joints (revolute, prismatic, fixed) and their limits.
- **Mass and Inertia**: Adding realistic mass and inertia values for each part.
- **Visual and Collision Properties**: Defining the appearance and collision bounds of each part.

The URDF is the foundation for simulating the robot's movement and interactions in the environment.

2. Import the Robot into a Simulation Environment

Using a physics-based simulator like **PyBullet**, the URDF file is loaded into a virtual environment. The simulator enables:

- Accurate physics-based interactions, including gravity, friction, and collisions.
- Testing and debugging the robot’s design in a safe, virtual space.

Verify the robot’s initial setup by manually applying forces or moving its joints to ensure proper articulation and stability.

3. Define the RL Framework

Set up the components needed for training with RL:

- **State Space**: Determine the information the robot will use for decision-making, such as joint angles, velocities, body orientation, and contact points with the ground.
- **Action Space**: Define the robot’s controllable outputs, such as torque, joint velocities, or specific joint angles.
- **Reward Function**: Design a reward function that guides the robot toward walking behavior. Rewards might include:
    - Positive rewards for forward movement, stability, and energy efficiency.
    - Penalties for falling, high energy usage, or erratic movements.

4. Choose and Implement an RL Algorithm

Select an RL algorithm suitable for the task. Common choices include:

- **Proximal Policy Optimization (PPO)**: A policy-gradient algorithm well-suited for continuous action spaces like robotic motion.
- **Deep Deterministic Policy Gradient (DDPG)**: Handles continuous actions effectively, useful for joint control.
- **Soft Actor-Critic (SAC)**: Balances exploration and exploitation for efficient training.

Implement the algorithm using libraries like **Stable-Baselines3** or custom implementations in TensorFlow or PyTorch.

5. Train in the Simulation

Begin the training process:

1. **Initialization**: The robot starts with a random policy (random movements).
2. **Exploration**: The agent tries different actions to understand their effects.
3. **Policy Updates**: The RL algorithm adjusts the policy to maximize cumulative rewards based on the agent's experiences.

The robot undergoes thousands of iterations, gradually learning to:

- Coordinate leg movements for forward motion.
- Maintain balance and stability.
- Adjust to minor perturbations in the environment.

6. Monitor Training Progress

Use tools to visualize and debug the training process:

- Plot reward trends over time to ensure steady improvements.
- Observe the robot's movements to check for unnatural behaviors or repetitive failures.
- Adjust hyperparameters (e.g., learning rate, exploration rate) if progress stagnates.

7. Test and Validate in Simulation

After the robot achieves stable walking, test its policy in diverse scenarios:

- Vary terrain conditions (e.g., slopes, uneven surfaces).
- Introduce disturbances (e.g., pushes or external forces).
- Test with changes in the robot's dynamics (e.g., payload variations).

This ensures the learned policy is robust and can generalize beyond the training environment.

8. Transfer to the Real Robot

Once the simulation training is successful, transfer the learned policy to the real robot:

1. **Calibrate Sensors and Actuators**: Ensure the robot's real-world sensors and actuators align with the simulation parameters.
2. **Test in a Controlled Environment**: Begin in a safe, controlled space to evaluate performance and identify discrepancies.
3. **Fine-Tune for Reality**: Address the "reality gap" by retraining or adjusting the policy to account for real-world factors like sensor noise, friction variations, and mechanical imperfections.
4.

By following this structured approach, reinforcement learning enables robots to autonomously learn effective walking behaviors, leveraging both simulation and real-world validation.

URDF Generation

A **URDF (Unified Robot Description Format)** is an XML-based file format used to describe the physical configuration of a robot. It defines the robot's structure, including links (rigid bodies), joints (connections between links), kinematic chains, and properties like mass, inertia, and collision geometries. URDFs are commonly used in robotics frameworks like ROS and simulation tools like Gazebo to model and simulate robotic systems.

Step 1: Prepare Your Model in Fusion 360

1. **Create or Import the Model**
   - Ensure your robot model is fully designed, including all parts and joints.
   - Organize components into **bodies** and **joints** corresponding to your robot's physical parts.
2. **Name the Components**
   - Assign meaningful names to each component (e.g., "Base", "Leg1_Coxa", "Leg1_Femur").
   - Proper naming ensures clarity in the URDF file.
3. **Define the Joints**
   - Use **Fusion 360's Joint feature** to define the relationship between components (e.g., revolute, prismatic).
   - Set appropriate joint limits, axes, and degrees of freedom for each joint.
4. **Check the Origin and Coordinate Frames**
   - Place the **origin** at the center of the robot base or a logical reference point.
   - Align the coordinate axes (X, Y, Z) with the intended motion and workspace of the robot.

Step 2: Install the URDF Exporter Plugin

1. Go to the **Fusion 360 App Store** or Autodesk's website.
   - Search for the **"URDF Exporter" plugin** (developed by Autodesk or a trusted third party).
2. Install the plugin by following the instructions provided.
   - Restart Fusion 360 after installation.

Step 3: Export the URDF

1. **Access the Plugin**
   - Open your robot model in Fusion 360.
   - Navigate to **Tools > Add-Ins > Scripts and Add-Ins**.
   - Locate and activate the **URDF Exporter** plugin.
2. **Configure Export Settings**
   - Specify the export directory where the URDF files will be saved.
   - Choose an appropriate format for the mesh files:
     - **STL**: Lightweight and widely supported.

        - **DAE (Collada)**: Provides better integration with some simulators like Gazebo.

3. **Set Up Joint and Link Parameters**
    - Verify that each joint and link in the robot model corresponds to a URDF element.
    - Check parameters such as:
        - Link masses and inertias.
        - Joint limits and effort/velocity properties.
4. **Generate URDF**
    - Click **Export** to generate the URDF and associated files (e.g., mesh files).
    - The exporter will create a folder containing:
        - The **URDF file**: Defines the structure and kinematics.
        - Mesh files for visualization and collision detection.

Step 4: Verify and Modify the URDF

1. **Open the URDF File**
    - Use a text editor (e.g., **VS Code**, **Notepad++**) to inspect the generated URDF.
2. **Check for Errors or Missing Information**
    - Verify joint names, types, and limits.
    - Ensure all meshes are correctly referenced with appropriate file paths.
3. **Add Missing Details** (Optional)
    - Include inertial properties if not generated automatically.
    - Add sensors, controllers, or other necessary parameters.

Step 5: Test in a Simulator

1. **Load the URDF in a Simulator**
    - Use tools like **Rviz**, **Gazebo**, or **PyBullet** to visualize and test the model.
2. **Verify Functionality**
    - Check for proper joint articulation and collision behaviors.
    - Debug any issues, such as incorrect link positioning or joint constraints.

Step 6: Refine and Iterate

- Modify the original Fusion 360 model or directly edit the URDF file to address any discrepancies.
- Repeat the export and testing process as needed.

Summary

This workflow allows you to create a robust and simulation-ready URDF from your Fusion 360 model, ensuring compatibility with robotics frameworks like ROS, PyBullet, or Gazebo.

RL Integration of Hexapod

Working closely with Dr. Hoan, I had the opportunity to witness firsthand how he leveraged the capabilities of PyBullet and the URDF model to bring our hexapod robot to life. The project began with the detailed creation of a URDF (Unified Robot Description Format) file, accurately defining the physical structure of our hexapod—the dimensions of its body, the joint configurations, and the characteristics of each leg segment. This URDF file was essential, as it served as the blueprint for the hexapod's virtual representation in the simulation environment.

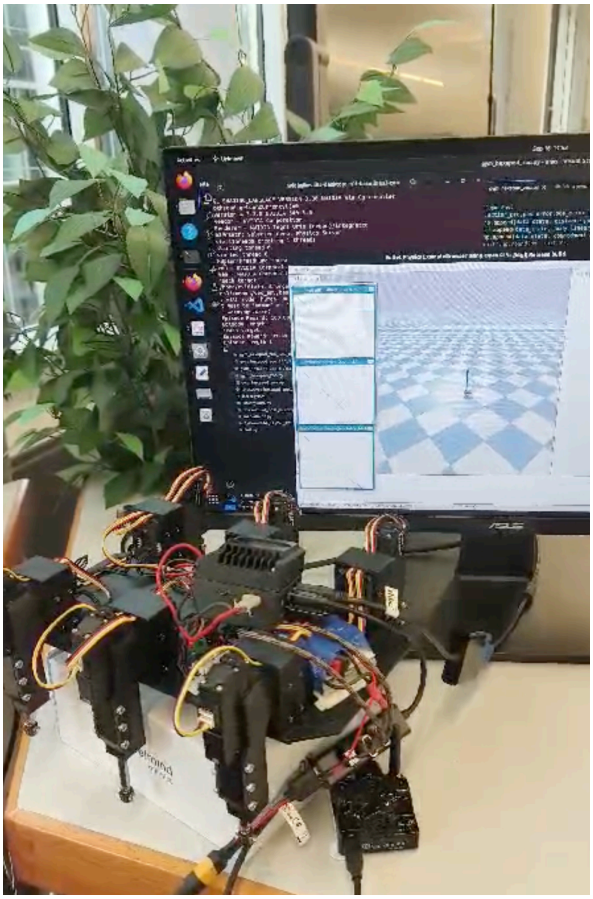


Figure 9: simulation and training of the hexapod

Once the URDF was ready, he utilized PyBullet, a powerful physics engine, to import our hexapod model. PyBullet provided a realistic simulation environment, allowing us to visualize how the hexapod would behave under different conditions without needing to experiment with a

physical prototype. His expertise in setting up the simulation was evident; he meticulously adjusted parameters such as joint friction, motor torques, and gravity to ensure the virtual hexapod closely mimicked real-world dynamics.

The next step was training the hexapod to walk. He employed a reinforcement learning algorithm within the PyBullet environment, guiding the hexapod to discover efficient walking strategies through trial and error. The algorithm would start by making random movements, receiving feedback based on the robot's performance—whether it moved forward, stayed stable, or toppled over. Over thousands of iterations, the algorithm gradually improved, refining its understanding of the complex coordination required between each of the hexapod's six legs.

During this phase, he monitored the progress closely, adjusting the reward functions and fine-tuning the simulation parameters to encourage more stable and energy-efficient movement. Each training session brought incremental improvements—small adjustments to the gait, more synchronized leg movements, and better balance on uneven terrain. Watching the simulated hexapod evolve from hesitant, uncoordinated motions to a fluid, stable walk was a remarkable experience.

Through his careful guidance and expertise in simulation dynamics, our hexapod transformed from a static digital model into a virtual machine capable of navigating complex environments

This link contains the video of it moving:

https://drive.google.com/file/d/1OcIqWl0NE_7KXcztKK8BvIzInhvjcD1Z/view?usp=sharing

Sim to Real Transition

In this section we will talk about how I assisted in the transition from the simulation of the hexapod to the actual hexapod:

Step 1: Training the Motion Plan in Simulation

1. **Simulate the Robot**
   - Used **PyBullet** to load the URDF of the hexapod and simulate its behavior.
   - Trained a motion plan using reinforcement learning to achieve locomotion.
2. **Test Policy Robustness**
   - Tested the motion plan extensively in simulation to ensure stability and adaptability under varied conditions.

Step 2: Creating Python Classes to Interface with Hardware

1. **Motor Control Classes**
   - Developed Python classes to interface with the physical motors.
   - These classes:
     - Translated joint position commands from the simulation into motor commands.
     - Sent the desired joint angles or velocities to the corresponding motor.
2. **Integration with Hardware Protocols**
   - Configured the classes to communicate with motor controllers via the appropriate protocol (e.g., PWM, CAN, I2C, or UART).
   - Ensured that the commands matched the motor specifications, such as speed, torque, and limits.

Step 3: Mapping Simulation Commands to Physical Motors

1. **Joint Command Translation**
   - Retrieved joint angle commands from the simulated motion plan.
   - Used the motor control classes to forward these angles to the respective physical motors on the robot.
2. **Joint Synchronization**
   - Verified that the timing and movement of physical joints matched the simulated behavior.
3. **Testing Motor Movements**
   - Conducted step-by-step testing for each joint to confirm accuracy before executing the full motion plan.

Step 4: Incorporating Feedback Systems

1. **Using Marvelmind Indoor GPS for Localization**
   - Mounted the Marvelmind GPS beacon securely on the robot using a Velcro strip for ease of adjustment and stability.
   -

- Configured the GPS system to provide real-time feedback on:
  - **Position**: Global X, Y, Z coordinates.
  - **Velocity**: Speed and direction of the robot.
  - **Orientation**: Heading angle derived from the GPS system and supplementary sensors (if used).

2. **Feedback Integration**
   - Wrote Python scripts to retrieve localization data from the GPS system.
   - Used this data to:
     - Validate the robot's movement against the planned trajectory.
     - Fine-tune the control system if deviations were observed.

Step 5: Real-World Calibration

1. **Environment Adjustments**
   - Accounted for real-world conditions like floor friction, uneven terrain, and noise in sensor readings.
   - Made adjustments to motor commands based on calibration tests.
2. **Iterative Testing**
   - Ran the motion plan on the physical robot and observed its behavior.
   - Iteratively adjusted:
     - The motor control parameters for smooth operation.
     - The motion plan if real-world deviations were significant.

Step 6: Validation and Refinement

1. **System Debugging**
   - Used logs and visualization tools to monitor joint commands, motor responses, and GPS data.
   - Identified discrepancies between simulated and real-world behavior.
2. **Refinement**
   - Tuned the motor control logic to improve accuracy and reduce latency.
   - Enhanced feedback utilization for more precise motion control.

Step 7: Future Integration for Quadruped

1. **Adapt the Existing Workflow**
   - Plan to reuse the motion planning approach developed for the hexapod to simulate and train locomotion for the quadruped robot.
   - Extend the Python motor control classes to handle the quadruped's unique joint configurations.
2. **Sim-to-Real Transition**
   - Apply the same sim-to-real steps—command translation, feedback integration, and iterative testing—to the newly manufactured quadruped robot.

Errors commonly faced with Sim to Real Transition

1. Model Discrepancies
   - **Description**: Differences between the simulation model and the actual robot.
     - Examples: Incorrect mass, inertia, joint friction, or actuator dynamics.
   - **Impact**: The robot might behave differently than expected, leading to instability or failure in real-world tasks.

2. Environmental Mismatches
   - **Description**: The simulated environment may fail to capture real-world complexities.
     - Examples: Unexpected terrain variations, lighting changes, or environmental noise.
   - **Impact**: Policies trained in simulation might fail to generalize to the real-world environment.

3. Sensor and Actuator Noise
   - **Description**: Simulations often assume ideal sensor readings and actuator responses, while real-world devices experience noise, delays, and calibration errors.
     - Examples: GPS drift, camera distortion, motor backlash.
   - **Impact**: The robot may misinterpret state information or execute commands inaccurately.

4. Lack of Robustness in Trained Policies
   - **Description**: Policies optimized in simulation may overfit to the simulated environment and fail to generalize.
     - Examples: A walking policy trained on uniform friction surfaces might fail on slippery or uneven ground.
   - **Impact**: Poor performance or failure in unexpected real-world conditions.

5. Hardware Constraints
   - **Description**: Physical robots have practical constraints that simulations might not capture.
     - Examples: Battery limitations, overheating, or material wear and tear.
   - **Impact**: The robot may not sustain long-term operation or might require re-design of motion plans.

6. Communication Latency
   - **Description**: Delays in transmitting commands or receiving sensor data in real systems are not always modeled in simulation.
   - **Impact**: Real-time control and coordination can be disrupted, especially for fast movements or reactive tasks.

7. Safety Concerns
   - **Description**: Mistakes during the sim-to-real transition can cause physical damage to the robot or its surroundings.
     - Examples: A poorly tuned locomotion policy might cause a robot to fall or collide with objects.
   - **Impact**: Expensive repairs and downtime, as well as potential injury in collaborative environments.

Summary

Addressing these challenges requires meticulous calibration of simulations, incorporation of real-world data into training (e.g., via domain randomization or hybrid learning), and robust testing on physical robots. Proactively managing these issues can significantly improve the success of sim-to-real transitions.

Conclusion

This project has made substantial progress in developing an autonomous motion planning system for legged robots, utilizing reinforcement learning (RL) and advanced simulation techniques. The initial focus was on simulating a pre-existing hexapod model in PyBullet, where I successfully applied RL to train the robot to walk, mimicking real-world scenarios and refining the control strategies. This allowed the hexapod to learn efficient locomotion patterns within a simulated environment. Through this process, I gained a solid understanding of the challenges and opportunities within the intersection of machine learning, robotics, and simulation.

The next critical phase was transitioning the learned motion plans from simulation to reality, which involved a careful and methodical approach to bridge the gap between the idealized environment of the simulator and the physical hardware of the robot. I developed Python classes to control the motors of the physical robot, which translated the joint position commands generated by the RL model into actual motor commands for the physical motors. This allowed the hexapod to perform the motions learned in simulation. The integration of the **Marvelmind Indoor GPS system** played a crucial role in providing real-time feedback on the robot's position, velocity, and orientation. By using this system, I was able to accurately track and correct the robot's movements, ensuring that the motion plan was executed precisely as intended in the real world.

The use of **domain randomization** and feedback loops in real-time allowed me to continuously calibrate the system and fine-tune the motion plan for the hexapod. With each test and adjustment, I improved the robot's ability to adapt to real-world conditions, which can often differ significantly from simulations due to factors like terrain variability, sensor noise, and mechanical tolerances. This iterative process was critical in ensuring that the learned policies were robust enough to handle the imperfections and dynamic nature of real-world environments.

In parallel, I took on the challenge of designing and manufacturing a custom quadruped robot, expanding on the lessons learned from the hexapod project. The quadruped robot introduced new complexities in its kinematics and control, requiring further development of the motion planning framework and motor control systems. However, the experience and tools built during the hexapod phase have laid a strong foundation for efficiently adapting the RL-based motion planning approach to this new robot. The quadruped will benefit from the same strategies of **sim-to-real transfer** and feedback integration that were used with the hexapod, with adjustments made to account for the unique challenges posed by the quadruped's configuration.

In terms of what I have learned throughout the course of this project, I now have a deeper understanding of reinforcement learning in robotics, especially in the context of legged locomotion. I have also gained valuable experience in **sim-to-real** processes, where real-world constraints and hardware integration challenge even the most robust simulation models. Additionally, I've had the opportunity to design and implement hardware interfacing solutions, using feedback systems like the Marvelmind GPS for precise tracking and validation.

Looking ahead, the work done so far has laid a strong groundwork for future developments. The next step will involve refining the RL-based locomotion system for the quadruped, applying the methods developed during the hexapod phase, and further optimizing the hardware interfaces to handle more complex tasks. This project has not only advanced my technical skills but also demonstrated the potential for advanced machine learning techniques to push the boundaries of robotic locomotion, setting the stage for future research and practical applications in autonomous systems.

References

- ***Rotary servo:***
  - https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/
  - https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_wizard2/
  - https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_sdk/overview/
  - https://emanual.robotis.com/docs/en/parts/interface/u2d2/https://hitecrcd.co.jp/industrial/images/products/irrobot/pdf/IR-USB01-User-Manual_V0_16L27_Eng.pdf
- ***Indoor gps:***
- https://drive.google.com/file/d1DOW7JpsWYrwMCMyxpU92j2Zg1w2qUDRtview?usp=sharing
- ***Linear servo:***
  - https://mightyzap.com/en/digitalarchive6/?uid=332&mod=document&pageid=1
  - https://hitecrcd.co.jp/industrial/images/products/irrobot/pdf/mightyZAP-Manger-manual_V.1.3_ENG_19D29.pdf
  - https://mightyzap.com/en/digitalarchive4/?uid=314&mod=document&pageid=1
  - https://mightyzap.com/en/digitalarchive5/?uid=495&mod=document&pageid=1
  - https://hitecrcd.co.jp/industrial/images/products/irrobot/pdf/IR-USB01-User-Manual_V0_16L27_Eng.pdf
- **Training Repository for Hexapod**
  - https://github.com/DLR-RM/rl-baselines3-zoo?tab=readme-ov-file
  - https://rl-baselines3-zoo.readthedocs.io/