COMS 4020 Final Report
(SD15) Real Steel Team

Iowa State University of Science and Technology

Dylan Booth, Zach Campbell, Isaac Lo, Max Mayer-Mader, Maxim Popov

Dr. Bowen Weng[1], Dr. Simanta Mitra[2], Yonas Sium[3], Mohammed Rahman[3]

[1] Client
[2] Instructor
[3] Teaching Assistant

Project Requirements

In the 2011 hit movie Real Steel, a robot learned to box by mimicking the motions of a human. This work seeks to actualize this vision, or more specifically, to map a human user's arm motions to a humanoid robot such that the robot's fists are placed relative to its body where the user would expect them to be in a copycat experience.

The applications of a successful implementation are numerous. In the most straightforward case, robots capable of mimicking human behavior via direct mapping can accomplish difficult or dangerous tasks. For instance, a robot may lift a far too heavy box for the human body to bear. Or, a robot might perform practical tasks such as construction in dangerous environments.

The ability to mimic human motion also has indirect benefits, most strikingly as a stepping stone toward fully autonomous humanoid robots. Reinforcement learning is a powerful tool to develop autonomous robot abilities, but such learning is a slow, expensive, and complex process. Copycat learning, where the robot learns to move by directly mimicking human behavior, can act as a strong baseline to learn further autonomous motion independently. The applications of fully autonomous humanoid robots extend far beyond those listed above for directly mapped motion mimicking robots, and could significantly enhance human capabilities in a limitless number of possibilities.

Returning to our motion mapping problem, we seek to detect and map human motion onto the robot body regarding the position of the hands concerning the body as a whole, ignoring for the sake of this work other parts of the body. This process is to be performed without any external components except for a camera, which is part of the client's G1 Unitree robot. Physical constraints are imposed so that no non-visual trackers are used.

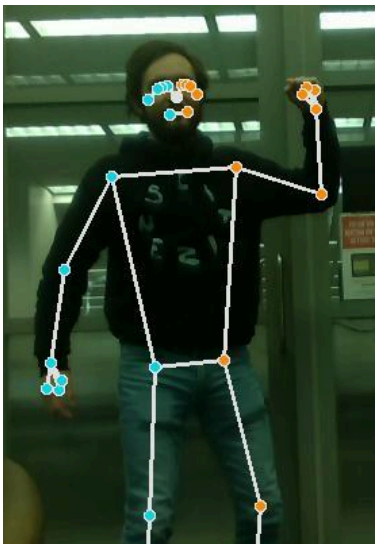
For the same reasons, all software must be capable of running directly on the robot. As such, we are constrained to use ROS1 and Python 3.8 to be capable of running on Ubuntu 20.04.

Additionally, we must ensure that all experimentation is conducted in a safe environment. Our robot is hung by its back from a harness connected to a stand behind it. A cord attaches the back of the robot's head to a computer. As such, all robot motion should be constrained not to move its hands behind its head or back. Furthermore, to not damage the robot, all motion must stay clear of colliding with itself. Due to imperfections in the robot's movement, a margin of error of 10 centimeters should be used for collisions. Lastly, all robot motion must be constrained to 1 radian per second

to easily halt motion and avoid damaging itself or anyone or anything in the immediate vicinity.

Project Design Overview

From a high level, there are two large pieces for a purely visual, human-imitating robot: a way to identify the pose of the human and translate that pose into its own body. With these two main components, there are sub-challenges required to overcome. These sub-challenges are mostly practical requirements for implementing the project well, for example, reducing motor-destroying point jitter in our solution. We will touch on these small and large design challenges and choices next.



MediaPipe Pose Output

Pose Detection

The D435i & MediaPipe

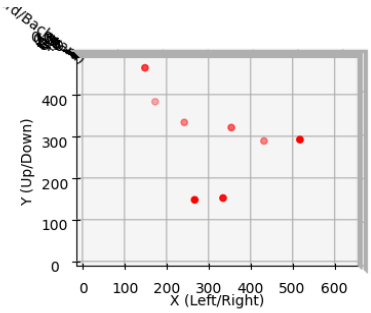
The input to the system is video frames from our client's **Intel RealSense D435i** depth camera. While the camera is active, it is continually polled for data using the PyRealsense library, a wrapper for Intel's C++ driver. These library calls produce two 640x480 2D arrays of data: an RGB pixel frame and a depth frame of distances in meters.

The color frame is passed into Google's **MediaPipe Pose**. This freely available pose detection library uses machine learning to produce a set of human body points, developed over the last five years by Google. MediaPipe has different dataset models that trade between speed and accuracy – "Lite", "Full", and "Heavy". For our purposes, we're shooting to publish a new instruction to the robot motors every 100 milliseconds, or at a rate of 10 Hz. This means we went with the "Full" model – it produces reasonably accurate point data and is also consistently able to process a frame in 40-60 milliseconds, well within our threshold. This gives us plenty of time to complete other preprocessing tasks and still publish at a rate greater than 10 Hz. Additionally, MediaPipe produces 33 landmarks, including lower body and facial points. However, we project this to eight relevant points – hands, shoulders, elbows, and hips.

One unique design challenge we faced was in attaching depth coordinates to the points produced from Google MediaPipe. This was because the real-time video detection is done through an asynchronous function that calls a callback with its results, so the depth frame can't be passed directly. To get around this, we initially tried embedding the

depth frame into the callback function using a closure (as Python functions can be defined and returned like any other variable). However, this was too slow as it required us to instantiate the pose detector on every frame. Instead, we used the timestamp associated with the frame (passed to the callback function by MediaPipe) as a key for a depth frame hashmap. As with most computer science problems, hash maps are magic and solve the problem.

Once MediaPipe detects the points' x and y pixel coordinates, each point is associated with that pixel's depth value. These points are then published on the /raw ROS topic and graphed using Matplotlib. This is helpful as it allows us to visualize what Mediapipe has produced.



Raw Point Data Graph

Point Preprocessing

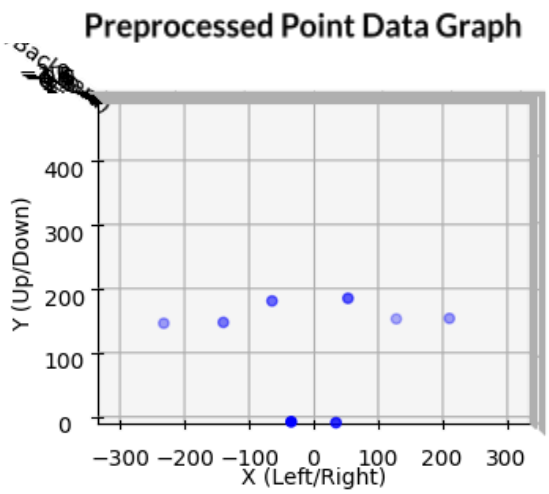
Once the XYZ point values are available, we perform several preprocessing steps. The first thing that happens on startup is a **calibration sequence**, in which the system collects point data for 5 seconds after it initially successfully detects a human pose. During this time, the human user holds a T-pose, with arms outstretched. This is to compute the distance from shoulder to hand, and shoulder to hip. These two values give us pixel-based vertical and horizontal scale factors, which we use at the end of preprocessing to convert the points to meters. These scale factors are computed separately for the left and right side of the body, and once calculated are stored and used until the system is restarted and recalibrated..

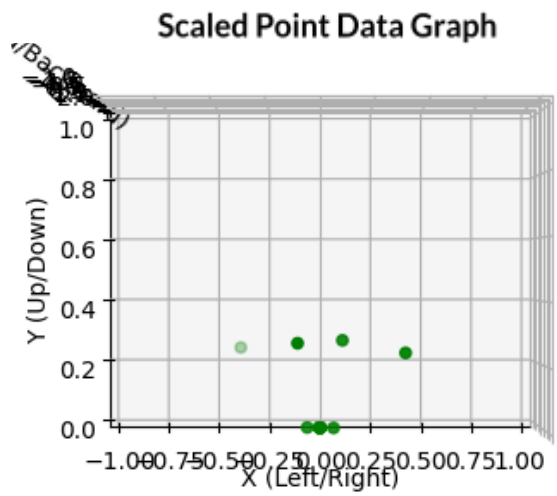
This sequence is best explained in a list of steps:

1. **Drop Bad Points**: Sometimes the camera fails to read a depth value for a given point, or MediaPipe fails to find it. We drop these null points or partial points and will fill them in later in the pipeline.
2. **Find Missing Points:** If a point isn't found for a frame, average the velocity of the previous 5 frames and use the last point's position to extrapolate and compute a point for this frame.
3. **Smooth Points:** Compare this point coordinates to the previous frame's point coordinates. If the distance is within a certain, adjustable threshold (currently 0.01 units) of the previous point, "snap" the point to that point's previous location, to avoid jitter between points that are very close to one another.
4. **Translation**: Our solution for computing the robot motor angles takes points in reference to the robot's pelvis (center of the hips). To prepare the points for this, we average the hip coordinates to get a pelvis value and then subtract this off of each of the points, making it the origin. This translates all points so they are positioned relative to the center of the hips.

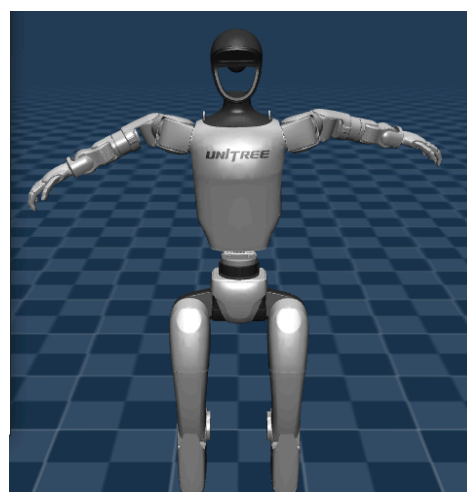
5. **Rolling Average**: In order to further reduce point jitter, for each point we compute a rolling average of the previous 5 frames' points, which further smoothes out jittery motion. These averaged points are kept separate from the real points that have been preprocessed thus far, so the average is not used in computing later points' rolling average value.
6. **Scaling**: The final step of preprocessing is to scale the points to convert from pixel measurements to meters. This is done using the scale factors computing during the startup calibration, and help proportion our points to the robot's body and size.

In the preprocessing phase, we have two more debug point graphs, one after the rolling average and one after the scaling step:





Inverse Kinematics



MuJoCo Simulator Motion

After preprocessing, the points are published on the /scaled ROS topic, and picked up by the /kinematics ROS node. This node handles calls to MoveIt, the inverse kinematics library, and will publish its results to the /robot_angles topic.

MoveIt "/compute_ik" Call

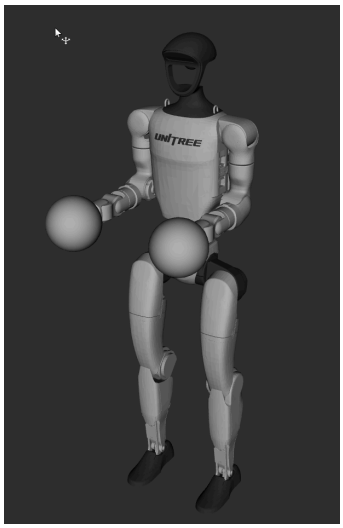
MoveIt is a library that provides powerful inverse kinematics solutions and motion planning integrated directly into ROS, a series of ROS nodes. However, MoveIt is significantly more capable than we need it to be – it will take a point elsewhere and produce a series of joint angles/instructions for getting the robot limb there. For our purposes, points are published every 100 milliseconds, which means the intermediate

planning motion MoveIt provides is not required. 10Hz is fast enough to be sequential instructions outright (there do exist speed limitations that are addressed in postprocessing; see section on Velocity Constraints below). As a result of this, we use the MoveIt ROS planning node, but only for a simple internal /compute_ik call, which takes in a target end effector (the hand) and a target location, and computes a series of motor angles that will place the hand at that position. This step is quite efficient to compute, less than 45 ms for both left and right hands (computed separately).

This calculation also considers whether or not a given point is valid. There are effectively two checks: can the robot's arm actually reach such a point (it may be too far away) and is that position going to result in a collision with anything in the robot's model file, (e.g. itself). If either check fails, we receive a "-31" error code, indicating that a valid angle set couldn't be found. If the checks are successful, we receive a "1" code, meaning we can continue with these computed angles.

G1 Unitree URDF

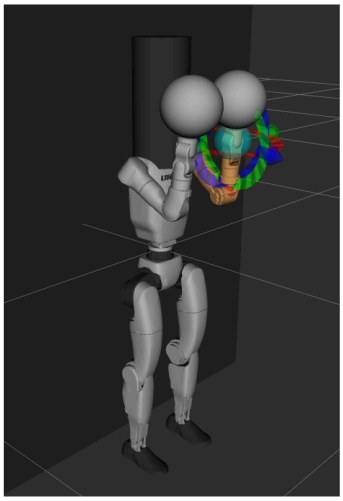
To accurately calculate arm joint solutions for the hand end-effector group, we required a similarly accurate model of the G1 Unitree robot. Unitree, the company, provides various models of the robot on GitHub. We chose a **29-dof full-body model** to leave the project opportunities for extension. This official URDF (Unified Robot Description Format) model made integrating MoveIt's inverse kinematics solutions straightforward. However, we did encounter various issues with the base model such as the hands lacking collision or the joint speed constraints being excessively lax.



G1 URDF Model

The lack of collision in the hands helped our team learn about URDF files and led to our discovery of a solution for the implementation of additional "safe zones" into our system.

From the first meeting our team had with Dr. Bowen, he emphasized the critical safety constraints in our project. One of his top priorities was protecting the body of the robot and the infrastructure around it (e.g. the rig it would hang from). We found adding more collider shapes and meshes directly into the robot model to be a very effective way to integrate these concerns into our inverse kinematics solutions.



URDF With Position Colliders

MuJoCo Simulator

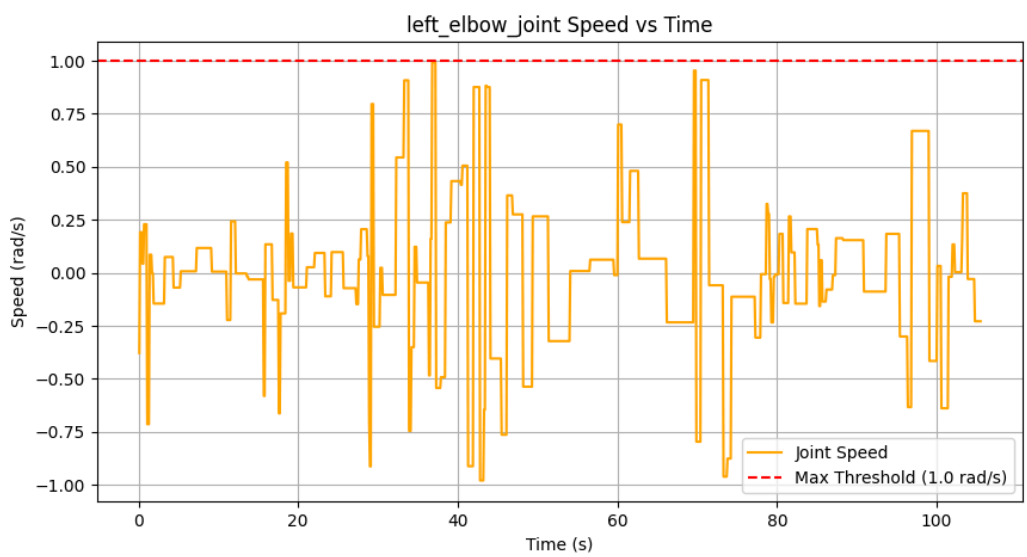
In order to debug and verify the accuracy of the computed motor angles, we implemented a ROS node to interface with the MuJoCo simulator. MuJoCo is a physics simulation system that can take our URDF and the angle instructions and arrange the robot to match those instructions. The simulator node we built simply calls the MuJoCo API functions with the output angles in real time. This simulator was the primary way we were able to safely end-to-end test the system without access to the robot itself.

Postprocessing and System Output

CSV and Graph Output

The end of our system is the robot_csv ROS node. This node receives the robot motor positions, conducts and saves the results of postprocessing, and after shutdown writes the results to a CSV file. Each row represents a frame of instruction, with a timestamp as the first value and then 12 columns of motor angle data. The system does not use the robot wrist motors, however, so four of those columns (two for each side) are all zeroes to be ignored.

Additionally, an output Dr. Bowen requested that we produce two graphs, position and velocity, for each CSV file. To achieve this, we wrote a separate script that takes in a completed CSV file and produces these graphs.



However, before these graphs and the final output we could send to our client, we had two more postprocessing steps, described in the following section.

Velocity Constraints

Alarmingly, the rotational velocity constraint listed for the URDF joints was around 30-36 radians per second. We consulted our client and found that while technically that speed limit is accurate, for safety and stability it would be unwise to come anywhere near it. We found a max speed of 1 rads/s to be sufficient in our final demo of the system.

To achieve this constraint of 1 rad/s, the joint positions are interpolated to ensure that the velocity never exceeds 1 rad/s, or 0.1 rad per frame update. For every joint, every frame is compared to the frame before it; if the difference between the two angles is greater than 0.1 radians, additional frames are inserted into the output so that the destination is the same but more time is taken to reach it, thereby reducing the speed of the motion.

Because each frame has eight motor positions, this interpolation calculation is done for all eight and then the maximum number of interpolated frames is what is actually interpolated. Once the number of frames is known, creating each frame consists of taking the difference between each joint's current position and its new position, and dividing it up across all the interpolated frames.

Consider an example: if the left elbow motor position is 0.4 radians and the previous elbow position was 0.1 radians, the system will compute that it needs 2 extra frames to remain under the speed limit. Instead of just adding the frame for 0.4 radians, it would add frames for 0.2, 0.3, and 0.4 radians to the output. Now imagine that the left elbow is the only joint that exceeded the speed limit, but the left shoulder yaw motor was supposed to move between 0.1 radians and 0.16 radians in that same single step. Instead of just having that motion happen immediately on the first interpolated frame, it is smoothly spread across all four new frames, resulting in instructions 0.12, 0.14, and 0.16 radians:

Time	Left Elbow	Left Shoulder Yaw	(other joints…)
0.0	0.1	0.1	…
0.1	0.2	0.12	(interpolated frame)
0.2	0.3	0.14	(interpolated frame)
0.3	0.4	0.16	…

A challenge we faced and had to fix for velocity was to drop the queue of interpolated frames if newly-received frames match up to the start. This handles the case where the human moves too quickly and then stops. Once the robot's joints are within a

speed-limit distance of the most recent joints, the queue can be cleared and the robot returns to matching the human's movement in real-time.

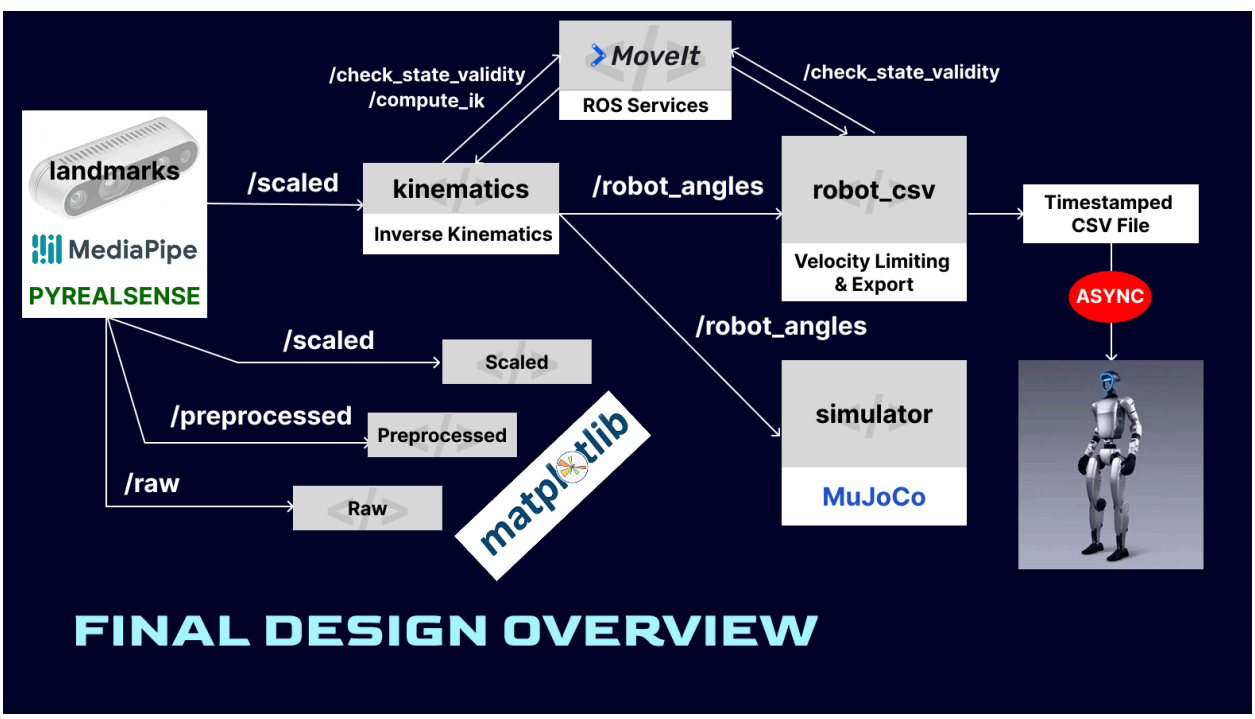
Pose Checking

After limiting the robot motor speed, it is necessary to "double check" the collision constraint. When interpolating frames, we are creating new angle instructions that were not produced by MoveIt. As an example, if the user quickly moves their hands near one another, the points given to MoveIt and the motor angles generated may not result in any collisions. However, slowing the hands down and interpolated frames may introduce collisions with one another. It is this case, and also some experience flakiness with MoveIt's initial collision checking, that necessitate a secondary, state validation call to the MoveIt ROS node.

To verify this, we use the MoveIt "/check_state_validity" service, which takes in a set of angles and uses the configured URDF to validate there are no collisions or violated constraints if the robot were to use the given angles. This check is efficient, and serves as a final safety measure before we publish the angles to the robot.

Project Overview Diagram

The following is an important diagram we used to understand and explain the flow of information for our project. Boxes represent ROS nodes and arrows represent topics or services. Production nodes that would be run in full usage are on top while debug nodes for visualization are on the bottom.



Individual Work Completed

Dylan Booth

- Put together ROS 1 documentation for team
- Configured custom ROS messages for Pose Coordinates & Angles
- Actively participated in designing of project Layout and how ROS could be used
- Researched and Learned about MuJoCo
- Implemented Basic MuJoCo Simulation
- Implemented ROS 1 into MuJoCo
- Implemented Point Interpolation into MuJoCo
- Implemented Speed Limiting into MuJoCo
- Implemented Position Validation Checker
- Tested all code with Depth Camera When needed

Zach Campbell

- Researched pose detection libraries and implemented MediaPipe calls
- Learned and implemented the PyRealSense library for interfacing with Dr. Bowen's provided Intel Realsense D435i depth camera
- Implemented graph_points ROS node for displaying 3D pose points in real time
- Implemented kinematics node using MoveIt inverse kinematics solution requests
- Extensive testing of inverse kinematics mapping, resulting in a scaling/transformation step to get accurate mapping and related wiki entry
- Designed point calibration step and figured out proportions to correctly request positions from the MoveIt solver
- Learned ROS1 structure and debugging tools
- Learned MediaPipe, PyRealSense, MoveIt, & MuJoCo APIs as used in the project
- Contributed to documentation in code comments & Gitlab wiki, as well as created slides for project presentations and wrote design sections for the final report

Isaac Lo

- Learned how ROS 1 works.
- Researched various simulation tools for our project.
- Researched and learned about MuJoCo.
- Researched and implemented universal robot description format robot models for our project.

- Researched Inverse Kinematics and various Inverse Kinematic solutions.
- Integrated the MoveIt motion planning library for Noetic.
- Extended our graphing to multiple graphing scripts for debugging point transformation.
- Implemented safe zone behind robot in the MoveIt inverse kinematic calculation.
- Implemented safe zone over the robot in the MoveIt inverse kinematic calculation.
- Implemented hand collision to the MoveIt calculation and the robot model.
- Implemented CSV pose validation checker.
- Implemented IK additional pose validation checker.
- Fixed collisions with inverse kinematics

Max Mayer-Mader

- Learned ROS1
- Created ROS launch files for running multiple nodes simultaneously
- Developed robot_csv.py to process robot joint angles with timestamps
- Implemented angle interpolation for handling motion speed limits
- Developed csv_plotter.py to track angle trajectory and speed to aid in debugging
- Learned and used MediaPipe for human pose detection
- Learned Linux for robotics development
- Set up velocity limiting constraints for safe robot operation
- Architected ci/cd pipeline for the repository
- Contributed to demo slides
- Made a README.html with project documentation and launch instructions for handover

Maxim Popov

- Consistently asked questions regarding requirements to ensure our solution remained on track to satisfy all of them
- Actively participated in the development of our architectural design of code flow
- Implemented the code-flow framework prior to the final iteration with the incorporation of the ROS framework
- Implemented 2d Mediapipe point visualization
- Implemented 3d Mediapipe point visualization
- Implemented point smoothing
- Implemented point interpolation
- Implemented the calibration stage
- Implemented statistical arm length calculation

- Integrated arm length calibration with mapping scaling
- Implemented z-scaling calibration
- Consistently ideated on potential failure points within our system and possible solutions to implement
- Created and presented the introduction and problem descriptions and values in demo slideshows
- Wrote the third chapter on requirements for the final report

Project Results Achieved

Technical Results

Our system detects and maps human motion onto the robotic body with a particular focus on hand positioning relative to the overall body. We've implemented a comprehensive pipeline including; visual input captured through an Intel RealSense D435i depth camera providing both color and depth information at 640×480 resolution. This data is processed using PyRealSense with timestamp-indexed hashmap storage for efficient retrieval.

Human pose detection utilizes Google's MediaPipe framework with its "Full" model configuration, maintaining a 0.95 confidence threshold to ensure reliable tracking. The system operates asynchronously, detecting 33 body landmarks (of which 6 will be used) that form the foundation of our mimicry system.

The raw coordinate data undergoes preprocessing, including:

- Initial calibration using a 5-second T-pose sequence to establish the user's proportions
- Signal processing with five-frame rolling averages to smooth noisy data
- Velocity extrapolation to fill missing tracking points
- Translation of coordinates relative to the pelvis
- Geometric scaling to convert pixel units to meters

Once preprocessed, our inverse kinematics implementation leverages MoveIt within the ROS ecosystem to translate hand coordinates into appropriate joint angles. This system handles successful positioning and, on failure, does not publish any data points when there are no safe data points.

The robot control system implements critical safety measures, including:

- Velocity constraints limiting movement to 1 radian per second with frame interpolation
- Position constraints that prevent the system from instructing the robot to reach behind itself
- Collision constraints that protect the robot from hitting itself with its arms or hands

Constraints and Safety Results

The project operates under several key constraints:

- All software must run directly on the robot using ROS1 and Python 3.8 on Ubuntu 20.04
- The robot is suspended from a harness connected to a stand behind it
- A safety margin of 10 centimeters is maintained for collision avoidance
- Movement speed is limited to 2 radians per second to ensure safety
- The system identified and rejected movements that would have caused collisions with the robot's support structure

Development Methodology and Achievements

The development process employed industry-standard software practices, including Kanban task management, systematic code review, and monthly sprint planning. Testing and ultimately our final demonstration utilized the MuJoCo simulation environment to avoid risking damage to the robot.

Key technical achievements include robust pose tracking despite intermittent sensor data loss, dynamic scaling for varying user distances, multi-layered safety systems, and seamless integration of complex frameworks including ROS, MediaPipe, MoveIt, and MuJoCo.

Project Real Steel demonstrates the viability of vision-only robotic mimicry systems, establishing a foundation for advanced human-robot interaction without specialized equipment. Ongoing refinements focus on kinematic optimization and comprehensive system documentation for end-users.

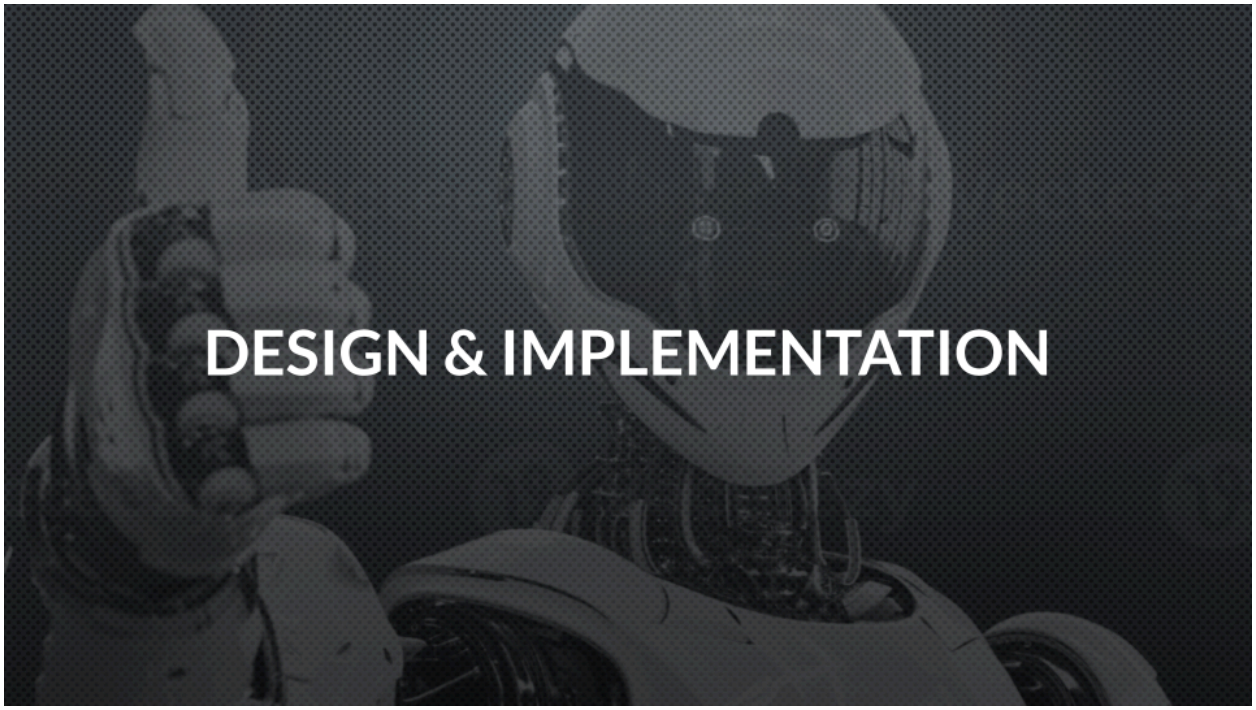
Appendix: Final Presentation Slideshow

PROJECT REAL STEEL

Dylan Booth, Zach Campbell, Isaac Lo,
Max Mayer-Mader, Maxim Popov

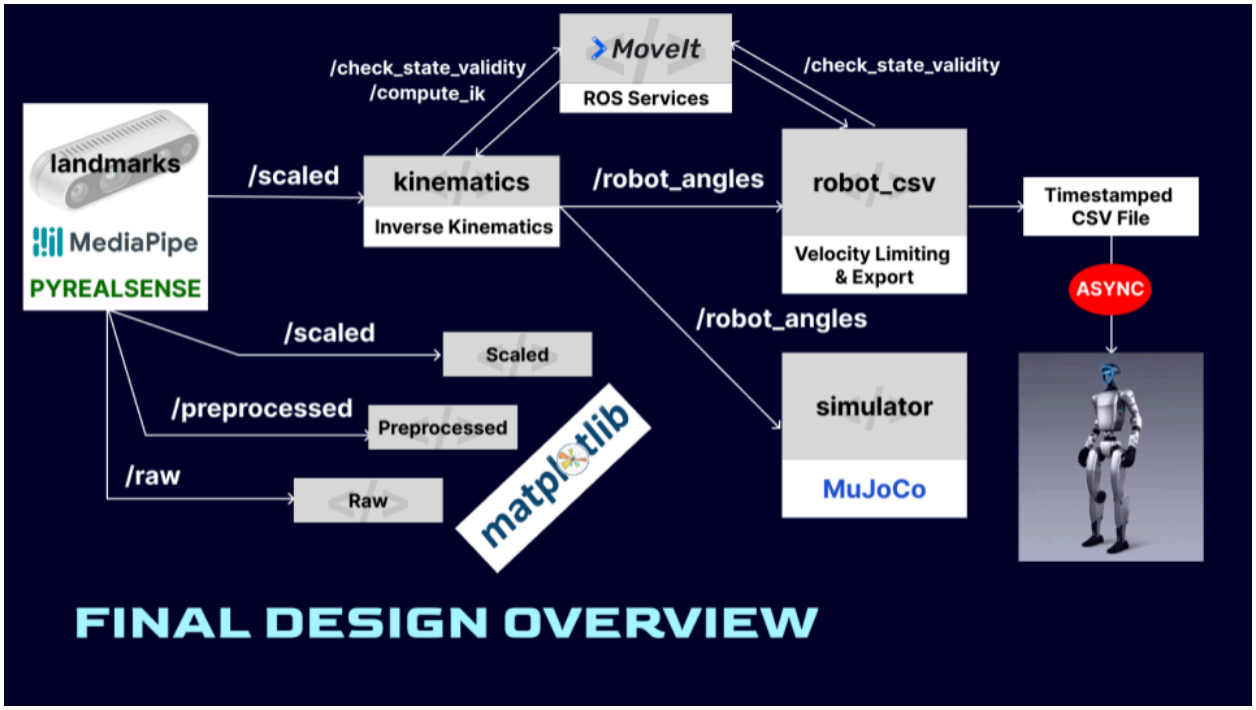
TOPIC

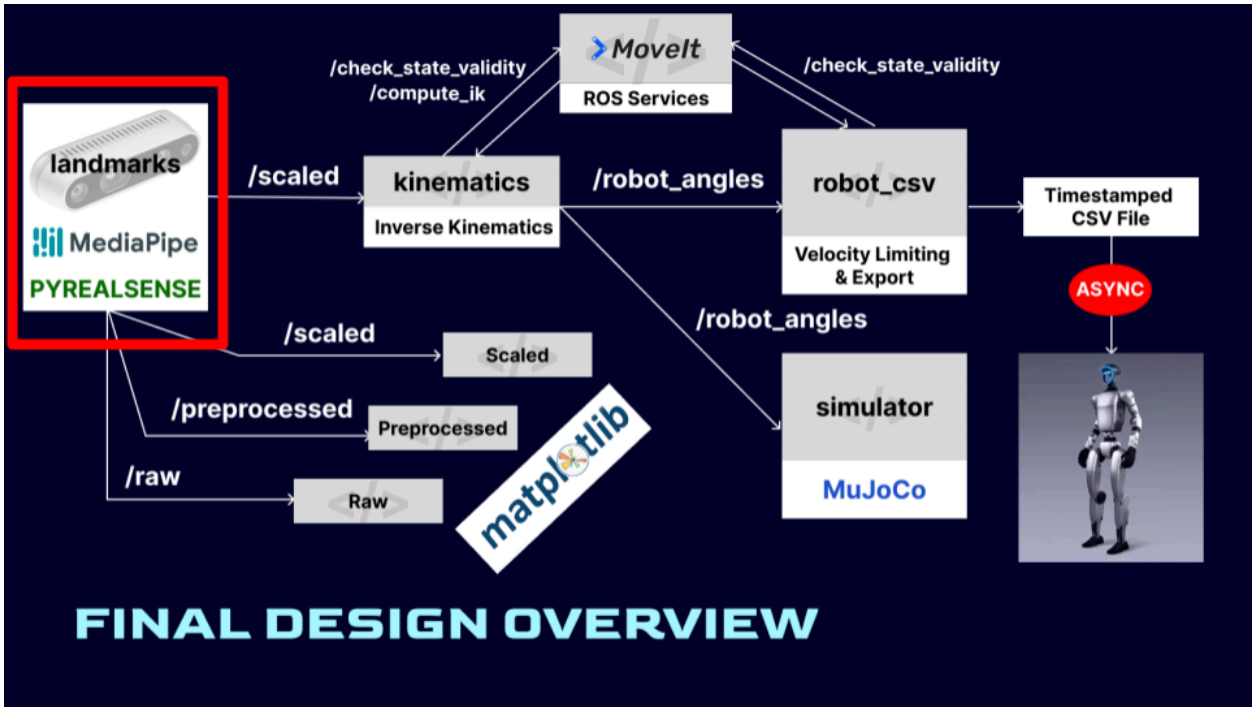
- Project Real Steel: A Boxing Robot
- Constraints
  - Visually: no on-body trackers
  - Hanging Orientation
  - Safety
- Applications
  - Difficult, Dangerous, or Menial Work
  - Copycat Learning



Libraries

- *Video Input:* **D435i Depth Cam; PyRealSense**
- *Pose Detection:* **Google MediaPipe**
- *Inverse Kinematics:* **MoveIt**
- *Graphing/Visualization:* **Matplotlib**
- *Robot Simulator:* **MuJoCo**
- *System Pipelining:* **ROS 1 Noetic**

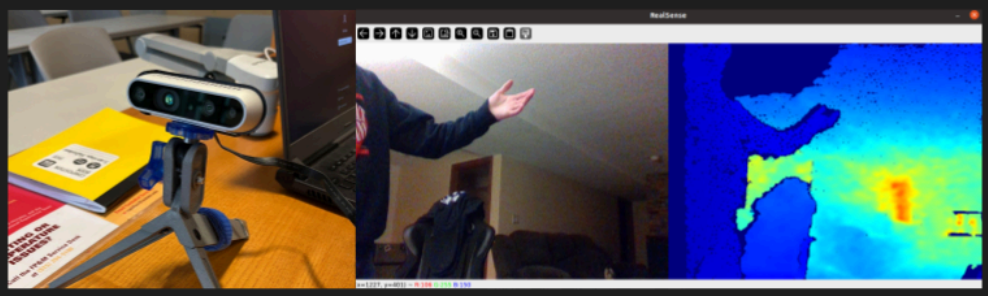

MoveIt
ROS Services
/check_state_validity
/compute_ik
/check_state_validity
landmarks
MediaPipe
PYREALSENSE
/scaled
kinematics
Inverse Kinematics
/robot_angles
robot_csv
Velocity Limiting & Export
Timestamped CSV File
ASYNC
/robot_angles
/scaled
Scaled
/preprocessed
Preprocessed
/raw
Raw
matplotlib
simulator
MuJoCo
FINAL DESIGN OVERVIEW


MoveIt
ROS Services
/check_state_validity
/compute_ik
/check_state_validity
landmarks
MediaPipe
PYREALSENSE
/scaled
kinematics
Inverse Kinematics
/robot_angles
robot_csv
Velocity Limiting & Export
Timestamped CSV File
ASYNC
/robot_angles
/scaled
Scaled
/preprocessed
Preprocessed
/raw
Raw
matplotlib
simulator
MuJoCo
FINAL DESIGN OVERVIEW

D435i Depth Camera

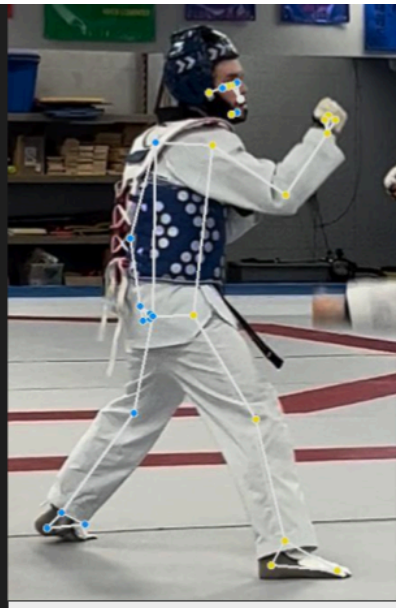
PYREALSENSE

- 640x480 Resolution
- Depth Frame with each image
- Store using hashmap for retrieval later
- PyRealSense Library

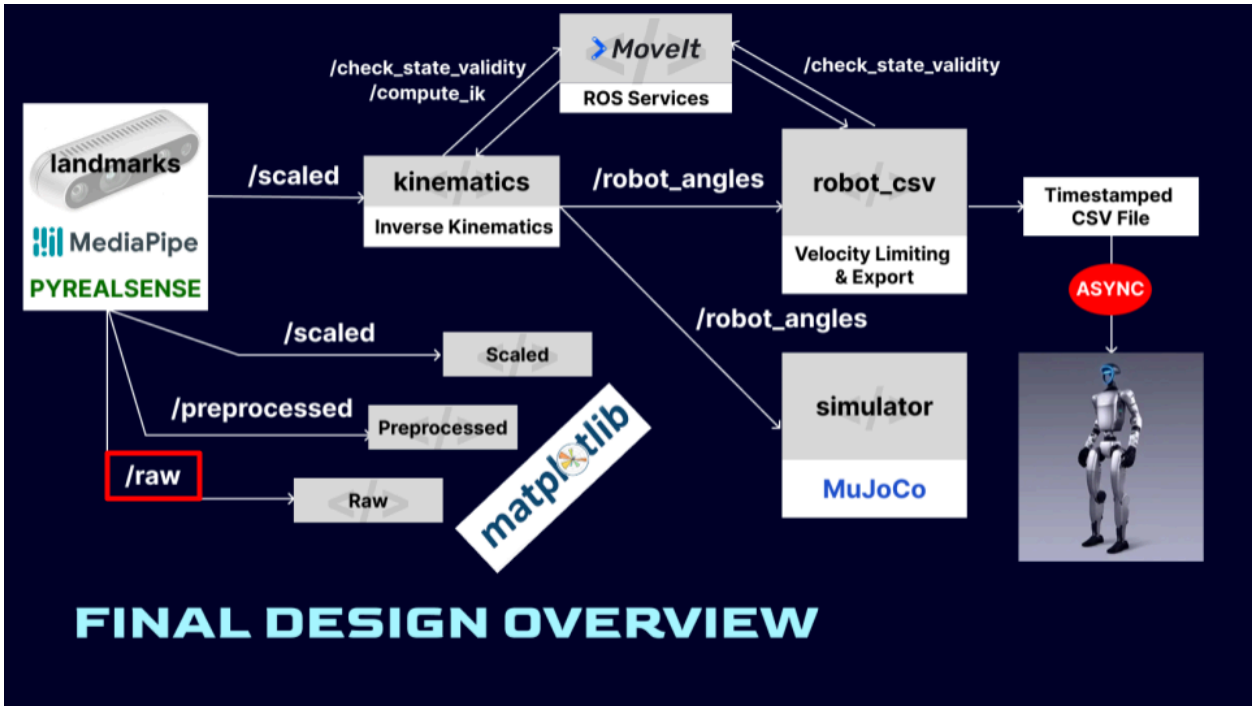


Mediapipe Pose Detection

- Developed by Google 2020-2023
- ML model "Full" to detect bodypoints from videos
- Asynchronously computes bodypoints, fires callback with results



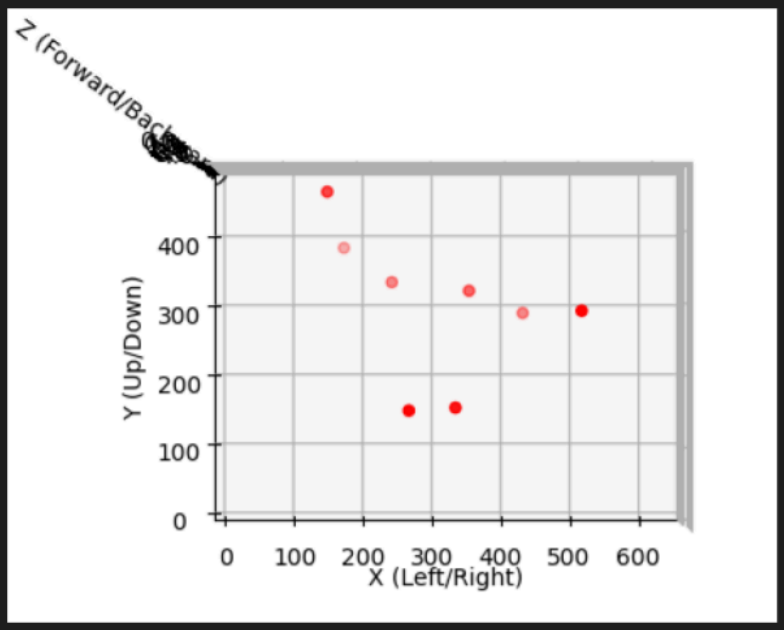
MediaPipe

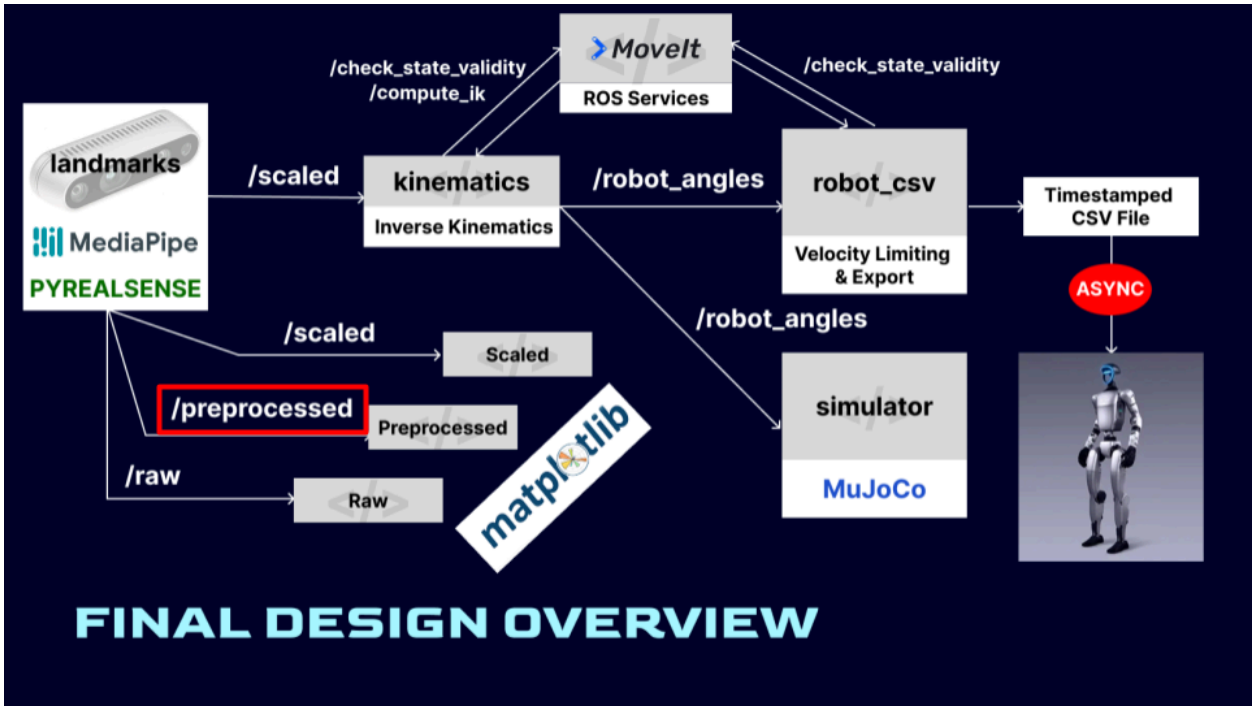


Raw Pose Coordinates

Bounds:
- X: [0, 640]
- Y: [0, 480]
- Z: [0, 4]

Raw point data from Mediapipe & Depth Camera



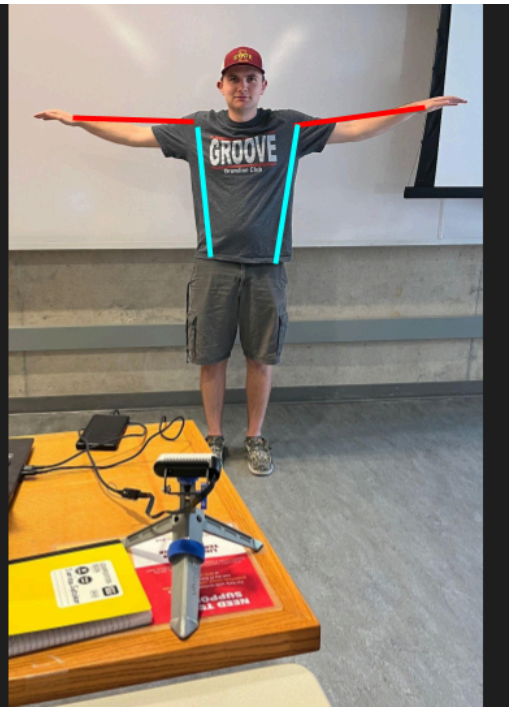


Preprocessing (1/3)

After first successful pose detection:

a. Wait 5 seconds for user to T-Pose
b. Record average shoulder-hand distance for 5 seconds
c. Save as arm length constant

These are saved & used later



Preprocessing (2/3)

- Velocity Extrapolation & Smoothing
  - For every frame, use average position & velocity of previous 5 frames to extrapolate & fill lost points
  - Ignore small variations in point data to reduce jitter
- Checks:
  - Depth camera drops z (depth) coordinate frequently
  - Cap velocity to prevent points from flying

Preprocessing (3/3)

Preprocessing: **Translation**

Compute pelvis – midpoint of the hip points

Subtract from all other points

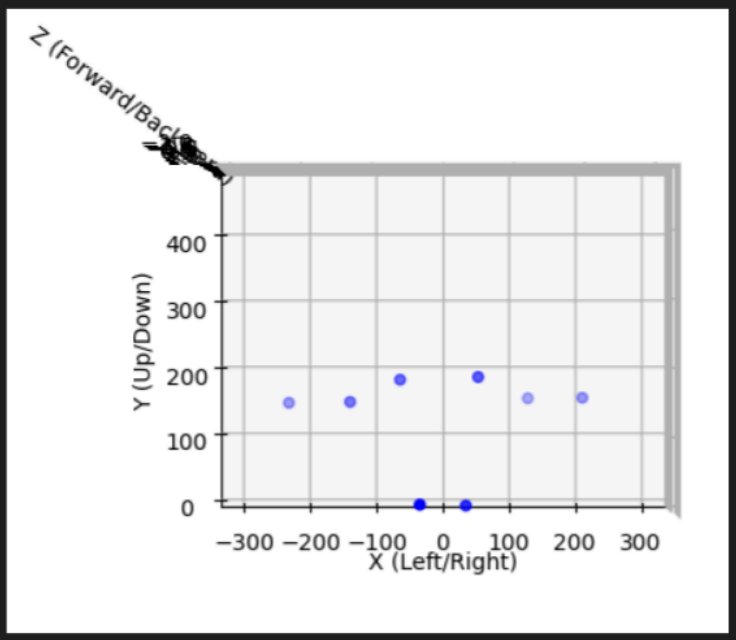
Makes all points relative to pelvis, as needed for IK

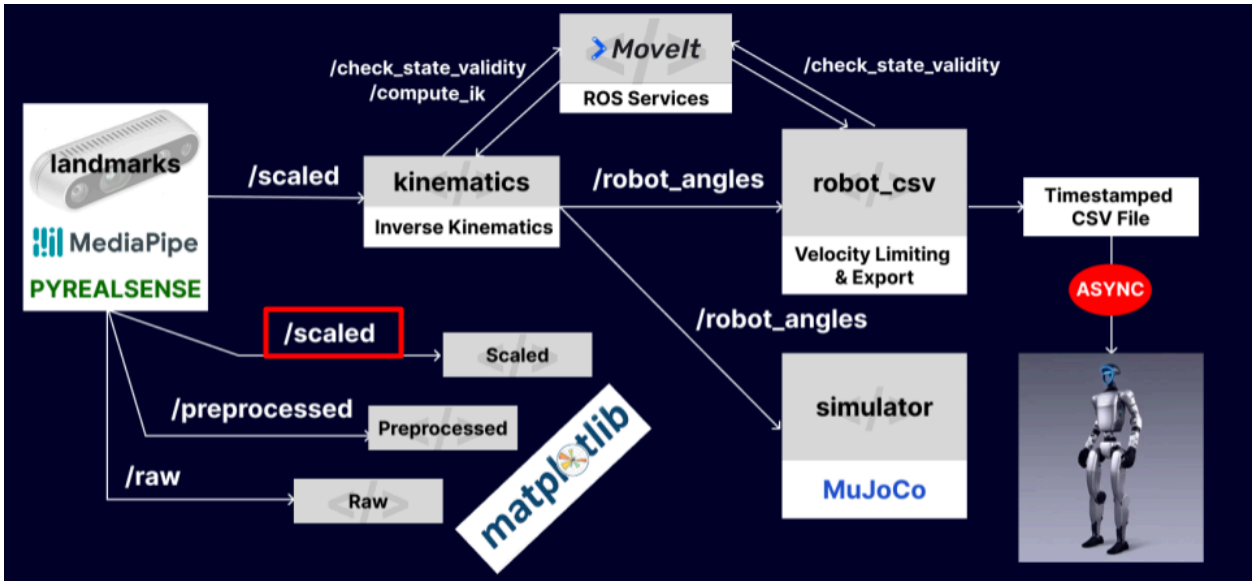
Preprocessed Coordinates

Bounds:

- X: [-320, 320]
- Y: [0, 480]
- Z: [-2, 2]

Interpolated, smoothed, and translated points



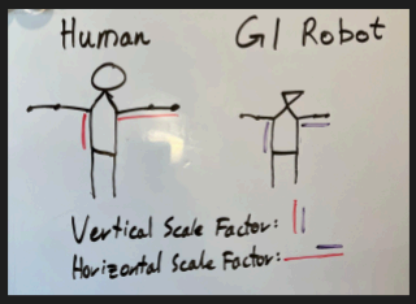


FINAL DESIGN OVERVIEW

Scaling (1/2)

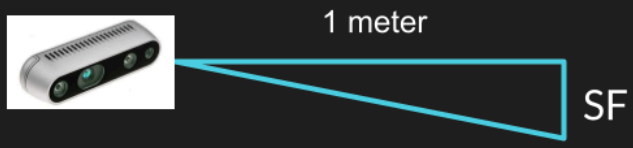
Need to scale from pixels -> meters for IK

- Use arm length for horizontal scaling
- Use shoulder-hip length for vertical scaling
- Depth coords are already in meters
- XYZ -> ZXY for robot orientation



Scaling (2/2)

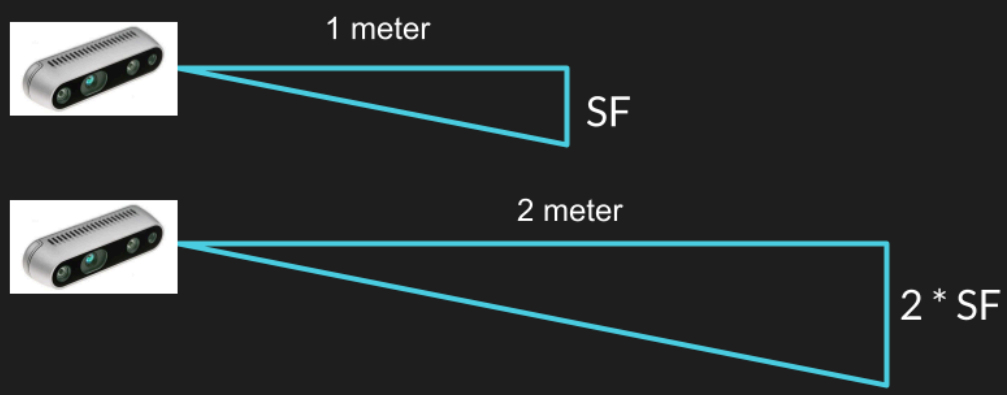
Use similar triangles to update scale factor across distance



Calibrate at 1 meter distance,
find length of arms (scale factor) SF

Scaling (2/2)

User changes distance, similar triangles gives new scale factor

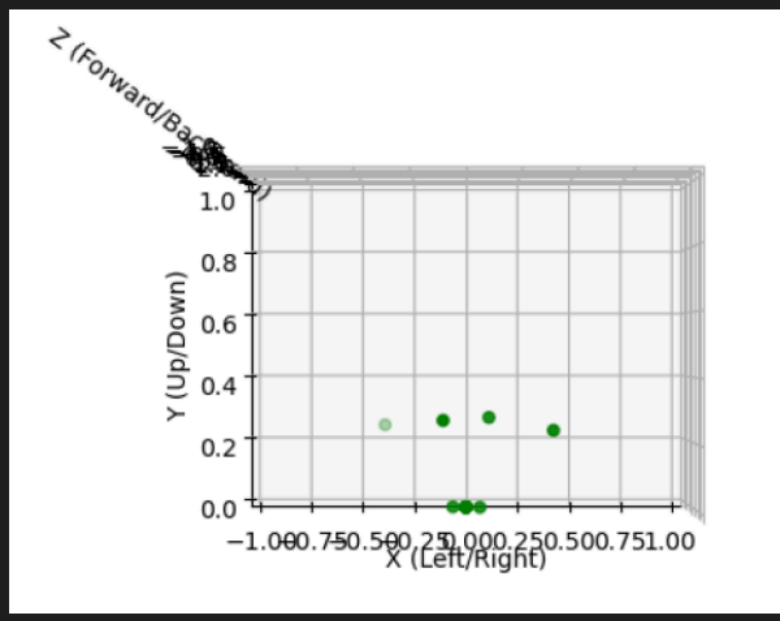


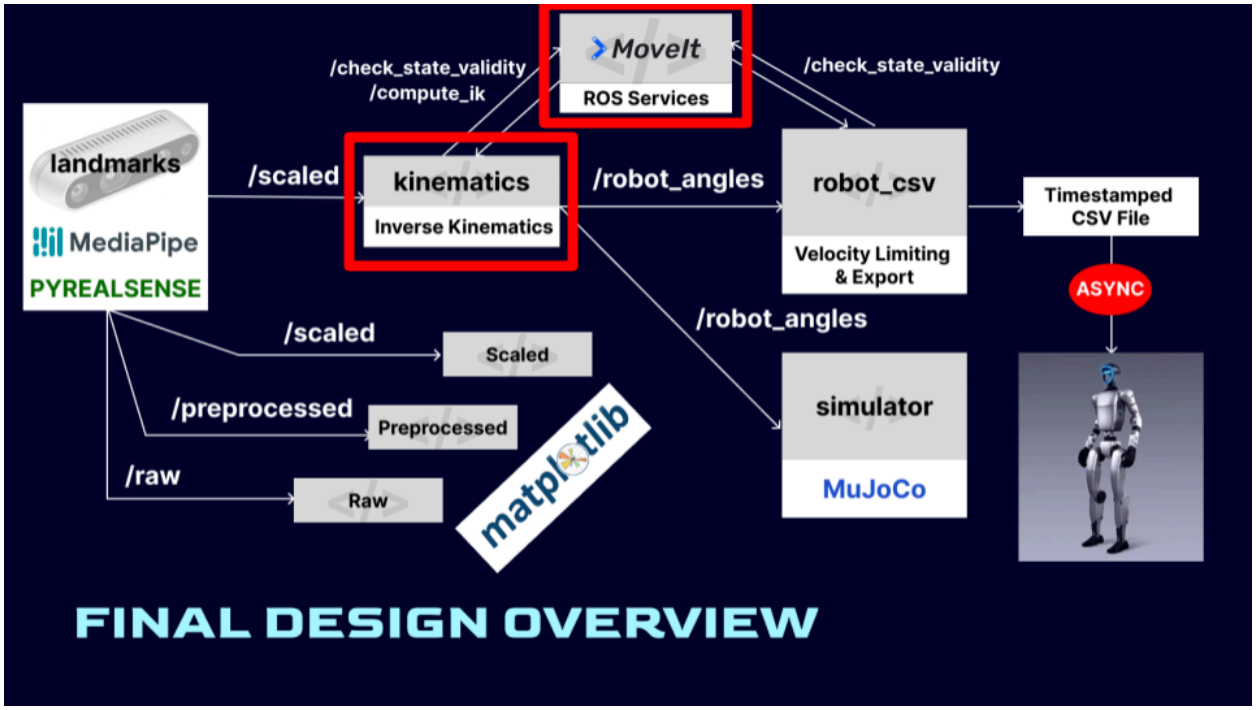
Scaled Coordinates

Bounds:

```
- X: [-1 to +1]
- Y: [0, 1]
- Z: [-2, 2]
```

Preprocessed points scaled to the robot's proportions for IK

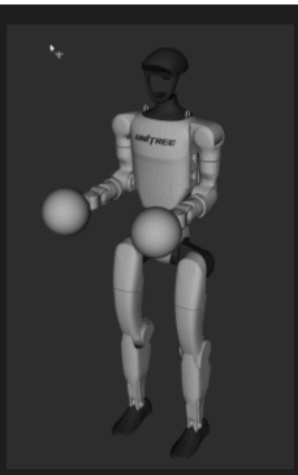




Kinematics Node (1/3)

MoveIt

- Receive /scaled points & request compute_ik service
- compute_ik is provided by MoveIt ROS nodes that are running concurrently to support computations
- Prepare an IK Request – joint group & target point

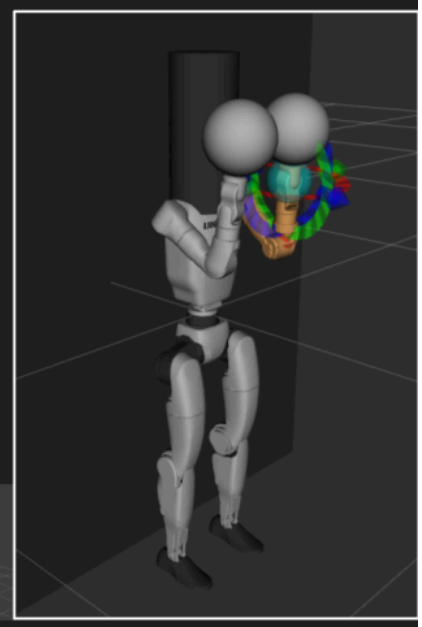


Kinematics Node (2/3)

- Configured with URDF File
- Base URDF is from Unitree
- Modified for additional constraints
- Allows for planning complex movements – more than we need

`/compute_ik`

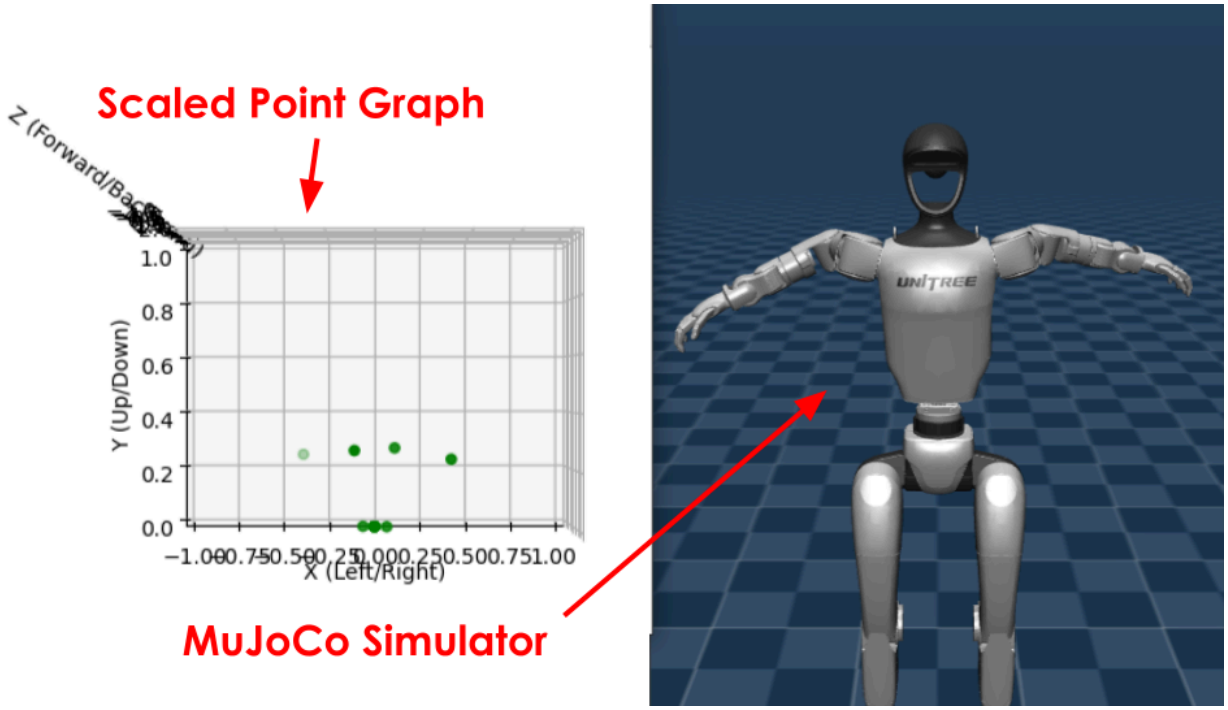
`/check_state_validity`

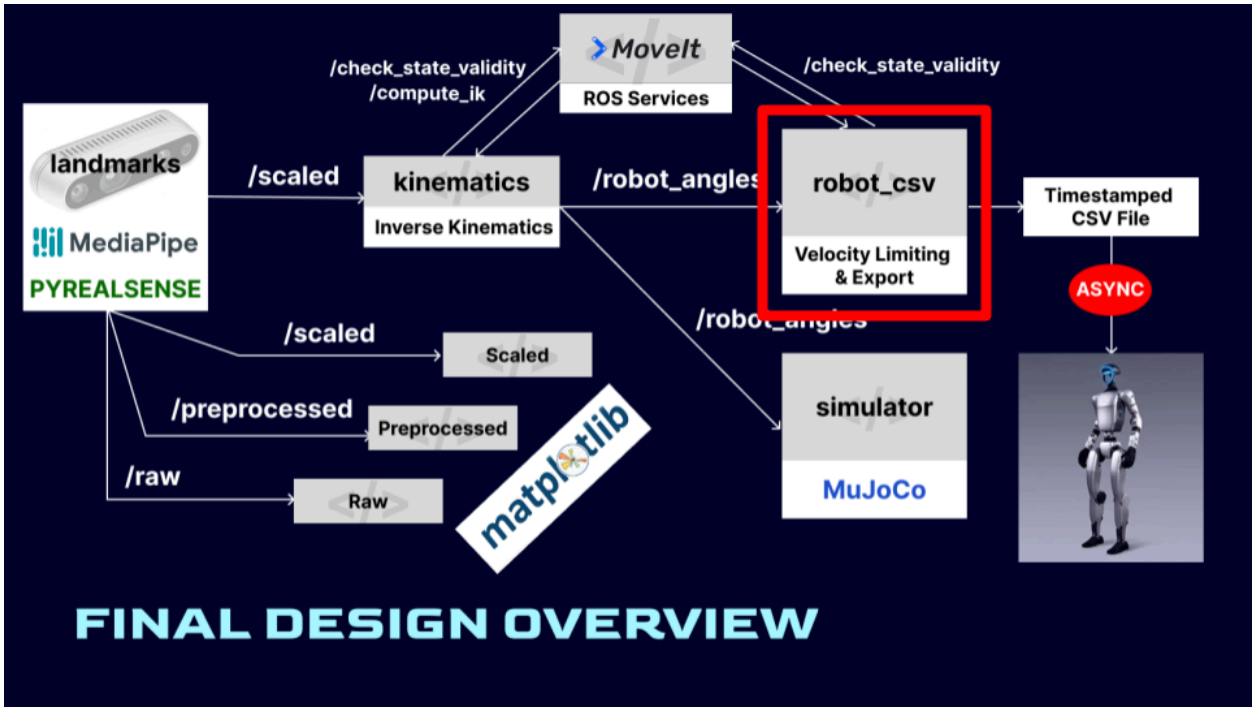


Kinematics Node (3/3)

- MoveIt handles positional restraints
  - Out of reach (too far away)
  - Invalid
- Receive errors for left and right hands separately

```
ROBOT_COORD_SYS LT: (0.02800000086426735, 0.14499999582767487, 0.05400000140070915) RT: (-0.017999999225139618, -0.12800000607967377, 0.03400000184774399)
CAMERA_COORD_SYS LT: (0.14499999582767487, 0.05400000140070915, 0.02800000086426735) RT: (-0.12800000607967377, 0.03400000184774399, -0.017999999225139618)
 -31, (-0.4519999921321869, -0.13600000739097595, 0.1289999932050705) | 1, (-0.03999999910593033, -0.1379999965429306, 0.03799999877810478)
 -31, (-0.49300000071525574, -0.18400000035762787, 0.1860000044107437) | 1, (-0.04100000113248825, -0.1420000046491623, 0.041999999433755875)
 -31, (-0.44699999690055847, -0.1899999976158142, 0.20600000023841858) | 1, (-0.04500000178813934, -0.15000000596046448, 0.041999999433755875)
```


Scaled Point Graph
Z (Forward/Back)
Y (Up/Down)
X (Left/Right)
1.0
0.8
0.6
0.4
0.2
0.0
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
MuJoCo Simulator
UNITREE


MoveIt
ROS Services
/check_state_validity
/compute_ik
/check_state_validity
landmarks
MediaPipe
PYREALSENSE
/scaled
kinematics
Inverse Kinematics
/robot_angles
robot_csv
Velocity Limiting & Export
Timestamped CSV File
ASYNC
/robot_angles
/scaled
Scaled
/preprocessed
Preprocessed
/raw
Raw
matplotlib
simulator
MuJoCo
FINAL DESIGN OVERVIEW

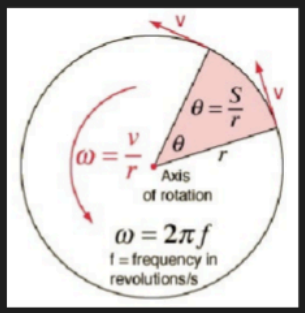
Robot_CSV Node

- Receives angles from /robot_angles and saves them to CSV file
  - timestamp, angle_0, angle_1, ...
- Timestamps are 10 Hz Apart
- Constraints
  - Velocity
  - Position

timestamp	left_joint_0	left_joint_1
0.000	0.8124596476554871	-0.039466213434934616
0.100	0.704815149307251	0.6107334494590759
0.200	0.5639855265617371	0.6929776668548584
0.300	0.5879011750221252	0.14779908955097198
0.400	0.3258069157600403	0.2050982564687729
0.500	0.23422592878341675	0.30206575989723206
0.600	0.173037588596344	0.44861915707588196
0.700	0.2861247658729553	0.6247268319129944
0.800	0.20040516555309296	0.3328632712364197

Design Challenge: Velocity Constraint

- Robot joints have speed limits
- Unitree Motor Limit of 35 radians/second
  - Approx. 5 RPS = 333 RPM
  - Unsafe speeds
  - Movements become imprecise
  - Requested/self-imposed **1 radian per second** limit



ANGULAR MOMENTUM

Angle Interpolation

- Problem: Human moves at > 1 radian/second
- Solution: Frame interpolation
  - Step every 0.1s; add more steps to get there
  - More steps = slower movement
  - If human stops moving, catch up & return to real-time

Computed Angles (IK)

Timestamp (seconds)	Motor Angle (degrees)
0.0	0
0.1	2
0.2	14

Interpolation Example

Speed Limit = 1 rad/s
= 0.1 rad/update
= 5.73 deg/update

Interpolation Example

Computed Angles (IK)

Timestamp (seconds)	Motor Angle (degrees)
0.0	0
0.1	2
0.2	14

14 - 2 = 12 deg/update

12 deg/update > 5.73 deg/update

12 / 5.73 = 2.094
= 2 extra frames

Computed Angles (IK)

Timestamp (seconds)	Motor Angle (degrees)
0.0	0
0.1	2
0.2	14

Output to Sim/CSV

Timestamp (seconds)	Motor Angle (degrees)
0.0	0
0.1	2
0.2	6
0.3	10
0.4	14

2 interpolated frames

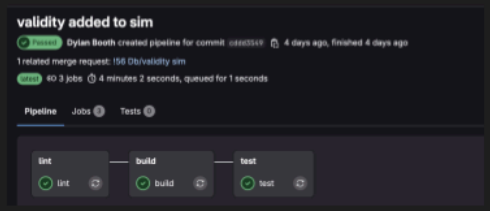
4 degrees/frame

Design Challenge: Position Constraints

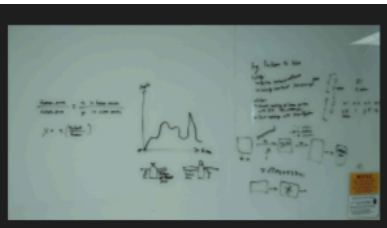
- MoveIt dynamic constraints are flaky
- Configuration issues
- Safety Check: `/check_state_validity`
- Validates hand/other collisions before publishing angles

Git & Code Review

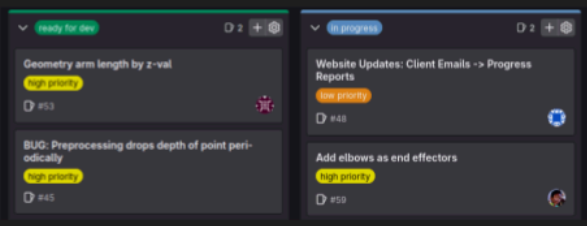
- Code Reviews
  - No work pushed to main directly
  - All work needed tested & read before approval to merge
  - Attempt to catch bugs prior to deployment to main
- Version Control & Branching
  - Feature/Fix/Experiment Branches
  - Source code management
- CI/CD pipeline



Issues & Task Division



- Sprint Planning
- Issue for each Feature or Bug in project
  - Associated branch per issue
- Issues labeled based on priority and progression
  - All members had >= 1 High Priority tasks
- Kanban Board
  - Progress visualization
- “Whiteboard Meetings”

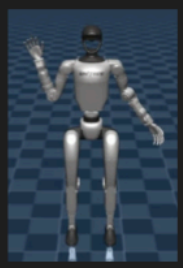


Dev Tools

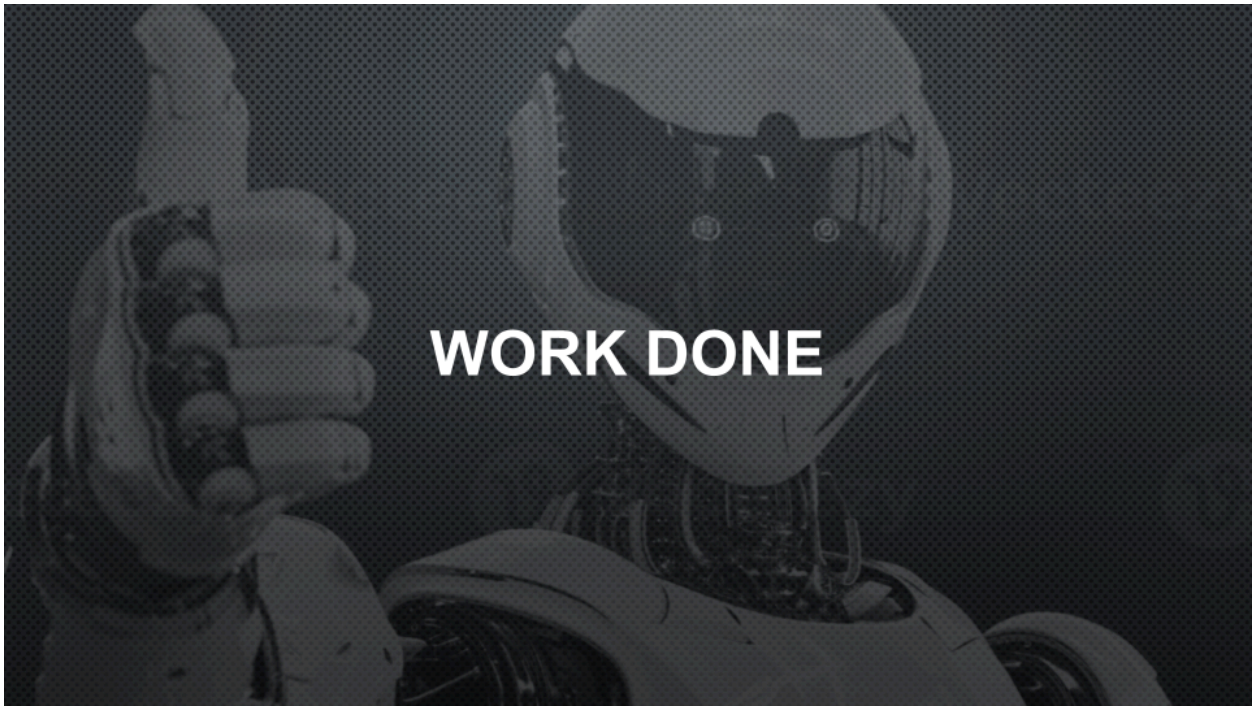
- ROS Launch Script
  - Start all ROS nodes simultaneously
  - Variants for debugging vs. “production”
- graph_points visualizer;
  - Raw/Preprocessing/Scaled graphs

```
<launch>
  <include file="$(find landmarks)/launch/landmarks.launch"/>
  <include file="$(find preprocessing)/launch/preprocessing.launch"/>
  <include file="$(find kinematics)/launch/kinematics.launch"/>
  <include file="$(find g1_moveit_config)/launch/demo.launch" output="log"/>
  <include file="$(find graph_points)/launch/graph_points.launch"/>
</launch>
```

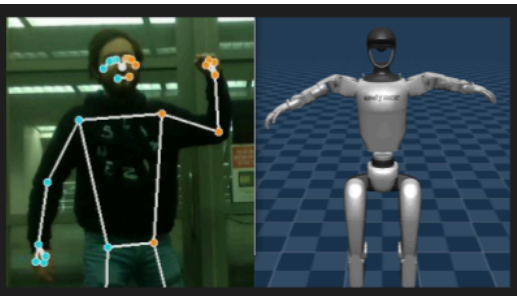
Dev Tools - Continued



- MuJoCo Physics Simulator
  - Mock testing for code
- ROS Bags
  - Recorded timestamp topics
  - Choose which topics you want recorded
  - Replays previously executed/processed data
- CSV replay node – double-check
  - Publishes whole CSV at 10 hz



Major Accomplishments



- 3D Pose detection
  - Smoothing & Scaling Data
  - Predicting pose coordinates
- Inverse Kinematics
  - (Reasonably) Accurate Positioning
  - Restraints on Users
  - Collision Avoidance
- Speed Limiting
  - Preserve Movements at Slower Speeds
  - Preventing damage to system

Minor Accomplishments

- Calibration
  - Accurate to human user, more reliable than just assuming
  - Geometry to scale as user moves forward/back
- Visualization
  - Graphing velocity of CSV data points

angles.csv 36.98 KiB

Edit Replace Delete

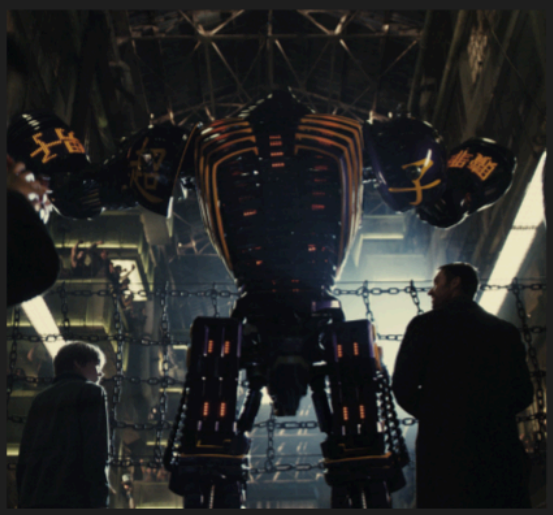
timestamp	left_joint_0	left_joint_1	left_joint_2	left_joint_3	left_joint_4	right_joint_0
0.000	0.8124596476554671	-0.03946621343493486	0.03344731032648358	0.5349076986312866	0.0	0.2897139191627502 4
0.100	0.7048151493072 51	0.6107234494590759	-0.7014712891307068	0.3209109902381897	0.0	0.3657128406929018
0.200	0.5639855326501737 1	0.6928776668548584	-0.8155505657196045	0.5663443803787231	0.0	0.3870844447612762 5
0.300	0.5679017502212 52	0.5477980895509798	0.3169603843624768	0.6367948247360 23	0.0	0.4739929245544433 6
0.400	0.3258069157600403	0.2050982564687729	0.8522373437987147	0.8933396802673334	0.0	0.4819705780862971
0.500	0.2342259267834675	0.3020657588723206	0.8887171532157698	1.0702506303767231	0.0	0.4095073342323303
0.600	0.173037588596344	0.4486191570758698	0.8301473855972 29	1.3057388259333838	0.0	0.1187976598739624
0.700	0.2861247658729553	0.6247268379129944	-0.4320448893503891	1.1396708488464355	0.0	0.1078259487898357
0.800	0.2004051655530929 6	0.3328832721364187	0.9235568642816272	1.1919147968292238	0.0	0.0928308706402778 6

Challenges

- Inverse Kinematics
  - Human-like motions are complicated
  - Not human-accurate motions can occur
- Collision Avoidance
  - Hands Clipping through Torso, Hips, and each other
  - Checking if interpolated points are valid
- Response Time

Project Extensions

- More accurate Inverse Kinematics
- Better collision avoidance
- Adding Locomotion (i.e. walking/moving around)
- Including Hands (e.g. making fist)
- Optimization



Summary Of Work

Dylan Booth
- ROS pipeline, MuJoCo Sim
- Extensive system testing

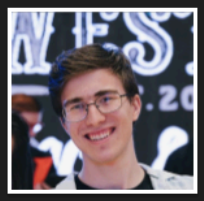
Zach Campbell
- MediaPipe Pose Detection, Depth Cam Interface
- Point Smoothing & Graphing, IK Calls

Isaac Lo
- Inverse Kinematics
- Robot Area Avoidance (e.g. head & back)

Summary Of Work

Max Mayer-Mader

- CSV Output Node, CI/CD Pipeline
- Velocity Limiting Interpolation + Verification Graphs



Maxim Popov

- Scale Calibration, Point Smoothing & Extrapolation
- MatPlotLib Graphing, Geometry Scaling