Software Requirements Document for Joseph Hoane

Team SD02

Authors: Junhyung Shim, Autrin Hakimi, Cal Hokanson-Fuchs, Thanh Mai, Nat Bui

1 Introduction

1.1 PURPOSE

Traditional chess engines exist in the digital realm, but integrating artificial intelligence with physical gameplay presents unique challenges. Our project addresses the gap between virtual chess engines and real-world play by creating an autonomous robotic system capable of physically manipulating chess pieces on a standard chessboard while playing at various skill levels against human opponents.

The purpose of this document is to establish how the robotic chess player system should interact with the end user, and to provide reference points to help other programmers to extend the system.

1.2 SCOPE

This SRS covers a well-established protocol to get the system running, as well as the system architecture and our solutions to inform the users on the intended uses of the system.

1.3 DEFINITIONS, ACRONYMS, ABBREVIATIONS

Term	Description
UR10e	An industrial robotic arm with 6 degrees of freedom. This is the main robot that allows the program to move the chess pieces.
Hand-e gripper	A gripper that allows the UR10e to pick up a chess piece.
ROS	Acronym for Robotic Operating System. It provides an interface to program the UR10e and the Hand-e gripper. Our system uses ROS Noetic.
Apriltag	A tag(similar to a QR code) used to detect the location of the piece in the image captured by a camera.
Stockfish	Chess engine that will solve the optimal move given the board configuration.
Inverse Kinematics / IK	Mathematical model to convert 3d coordinates to list of angles for the joints of robots. Transformation is solved using numerical methods.

1.4 REFERENCES

For more information regarding the system's core hardware and software, please consult the following:

- UR10-e: https://www.universal-robots.com/download/manuals-e-seriesur20ur30/user/ur10e/512/user-manual-ur10e-e-series-sw-512-english-international-en/
- Hand-e: https://assets.robotiq.com/website-assets/support_documents/document/Hand-E_Instruction_Manual_e-Series_PDF_20190122.pdf
- ROS: https://docs.ros.org/
- AprilTags: https://april.eecs.umich.edu/software/apriltag
- Stockfish: https://stockfishchess.org/

2 Overall Description

2.1 PRODUCT PERSPECTIVE

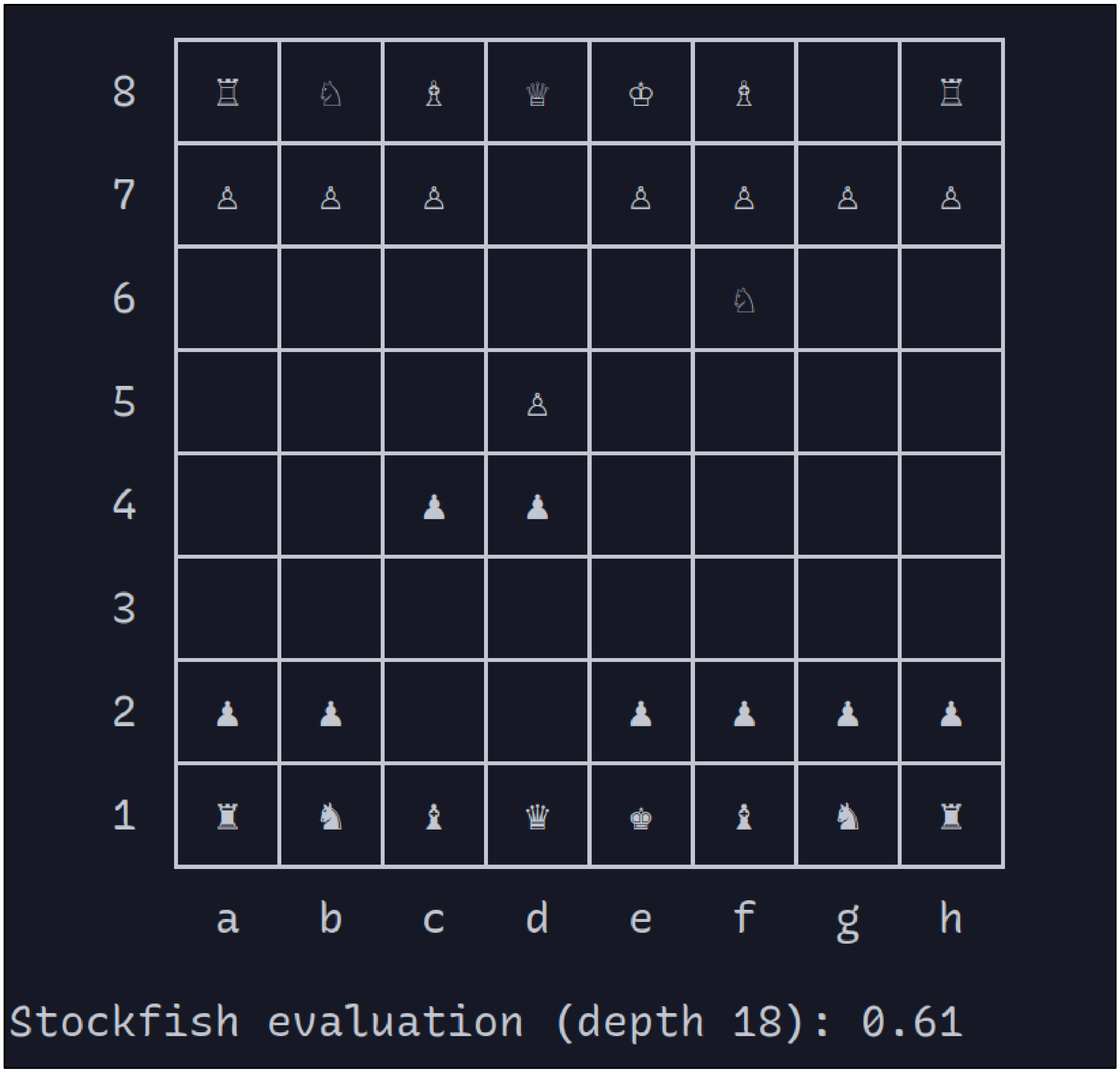
“Camera Module”, the OpenCV based program module that detects the chess pieces marked with April Tags
“Chess Module”, the wrapper class to query Stockfish
“Arm Control Module”, the program that controls the arm via ROS interface
“User Interface Module” the program that displays information to the user and controls the game.

2.1.1 Concept of Operations

This project leverages the AprilTag detection system, computer vision technology, and Stockfish to guide the UR10e robotic arm to play a game of chess autonomously in the real world.
The project consists of three core modules: Camera, Chess, and Arm Control.
The system is supported by ROS, which provides an interface to the robotic arm, allowing the software (written in python) to interact with the arm. With ROS, users can access information such as movement paths, error logs, and status of the robot and its joints.
*A few modifications to the code might be needed

2.1.2 User Interface

2.1.2.1 An example board state displayed to the user whenever a move is made.

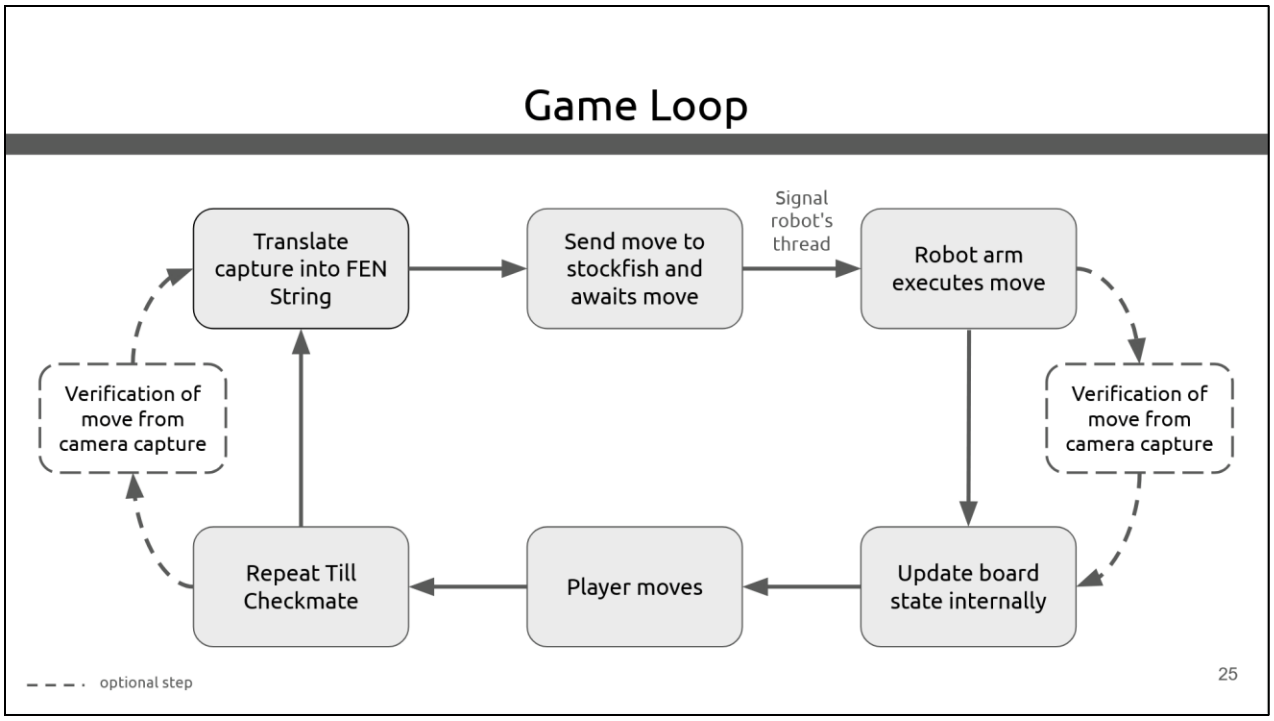


```
Stockfish evaluation (depth 18): 0.61
```

Other outputs and logs are omitted for simplicity.

2.1.3 Operations

Game Loop is visualized as below:


Game Loop
Translate capture into FEN String
Send move to stockfish and awaits move
Signal robot's thread
Robot arm executes move
Verification of move from camera capture
Verification of move from camera capture
Repeat Till Checkmate
Player moves
Update board state internally
optional step
25

2.2 PRODUCT FUNCTIONS

User plays a game of chess against the Robotic Chess Player.

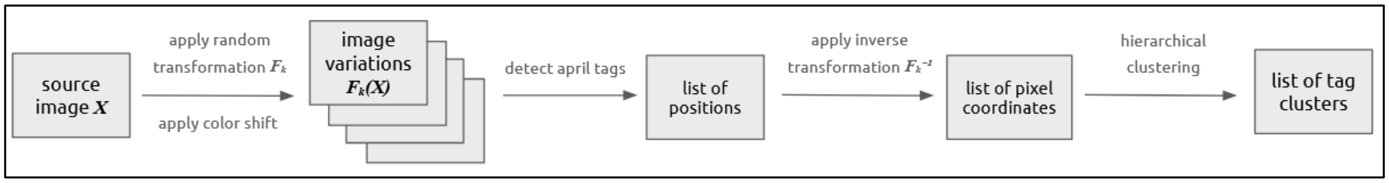
1) GameManager
   a) Grabs camera frames → AprilTag detections → constructs board state (FEN)
   b) Invokes Stockfish engine → returns best move in UCI format
2) ChessMovementController
   a) Gets the UCI strings (e.g. “e2e4”).
   b) Converts each move into a tiny “pick-and-place” motion plan:
   c) Pre-grasp above source square
   d) Descend, close gripper, lift
   e) Transit above destination
   f) Descend, open gripper, retract
   g) Packages those waypoints into ROS trajectory goals (FollowJointTrajectory and CommandRobotiqGripperAction)
3) RobotUR10eGripper (ROS node)
   a) Subscribes to /joint_states and /gripper_joint_states for real-time feedback
   b) Arm: uses a FollowJointTrajectoryAction client to drive the UR10e’s 6 joints
   c) Gripper: uses a CommandRobotiqGripperAction client to open/close the adaptive gripper

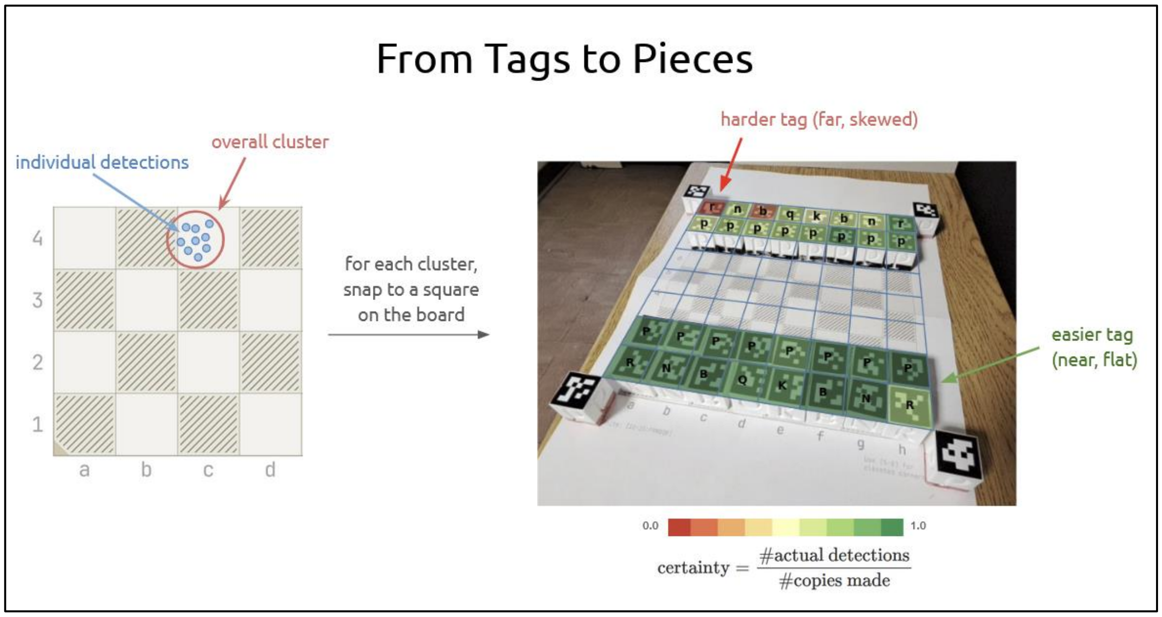
2.3 FEATURES

2.3.1 Vision System

The vision system uses Perturb-detect-clustering mechanism to detect April Tags even in poor lighting conditions. It will apply randomized affine transformations along with color perturbations to eventually detect the tags.

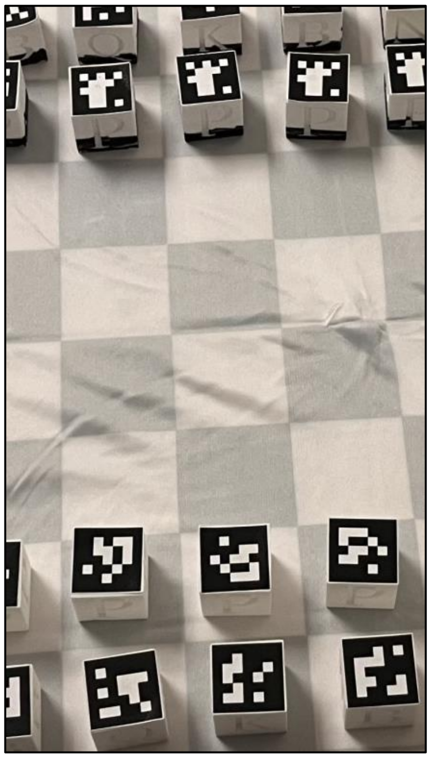
1. **Perturb**: Duplicate to many copies, apply random transformations and color shifts
   a. Transform: offset, rotate, scale, shear
   b. Color: brightness, contrast, gamma
2. **Detect:** Run apriltag detection upon each variational image
3. **Cluster:** Collect detections and cluster
   a. Transform component is inverted
   b. Apply single linkage, agglomerative hierarchical clustering


source image X
apply random transformation F_k
apply color shift
image variations $F_k(X)$
detect april tags
list of positions
apply inverse transformation F_k^{-1}
list of pixel coordinates
hierarchical clustering
list of tag clusters



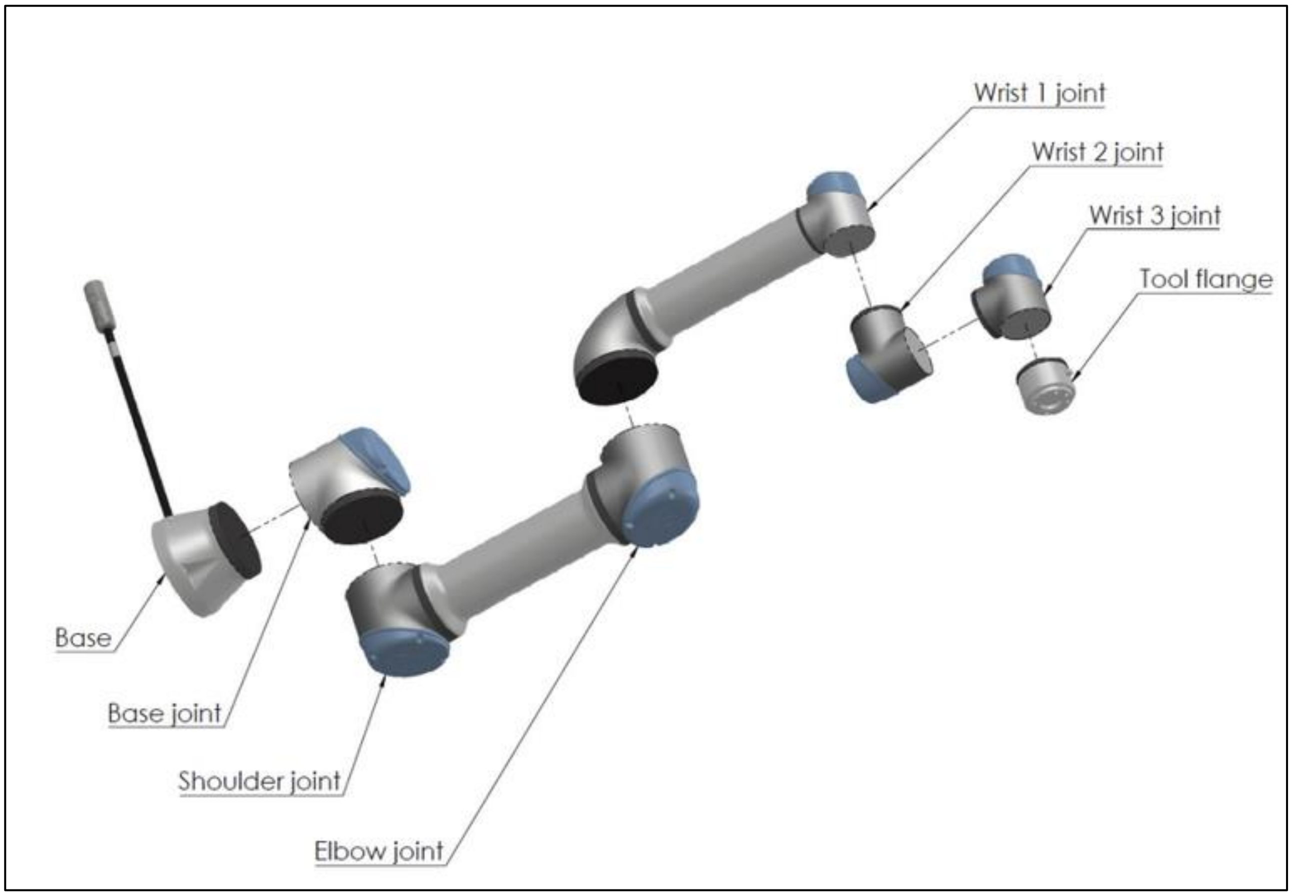
Pieces:

Each piece is a perfect cube, with an AprilTag on the top for detection and a letter (e.g. P for pawn, Q for queen) etched into each of the four sides. Black pieces (top of image) are denoted by the lower half of the piece shaded black



2.3.2 Robot Control System

2.3.2.1 Arm: Universal Robots UR10e



The Universal Robots UR10e is a 6 degrees of freedom arm, meaning it has six independently actuated joints that give full spatial orientation and reach:

- **shoulder_pan_joint** at the base to swivel the arm left and right
- **shoulder_lift_joint** to raise and lower the upper arm
- **elbow_joint** for forearm pitch
- **wrist_1_joint, wrist_2_joint,** and **wrist_3_joint**, which together orient the end-effector

We control these via ROS-Control's scaled_pos_joint_traj_controller, sending position goals to each axis. This six-axis freedom lets us position the gripper precisely over any square on the board."

2.3.2.2 Gripper: Robotiq Hand-E Adaptive Gripper

For grasping, we use the Robotiq Hand-E Adaptive Gripper. It has two opposing fingers that passively conform to a piece's shape. We don't have force sensors, so we drive it purely in position mode, open or closed, and we read back its joint state. In our code (jh1/robotics/robot_ur10e_gripper.py), we detect a successful grasp when the gripper closes but its reported position stops short of fully closed, indicating a chess piece is between the fingertips. This simple feedback loop makes our pick-and-place robust without extra sensors.

2.3.2.3 FollowJointTrajectory & gripper ActionGoals

FollowJointTrajectory and the “gripper ActionGoals” are just the two ROS-action interfaces we use to talk to the hardware.

- FollowJointTrajectoryAction
    a. This is a standard ROS action (in control_msgs) for commanding a time-parameterized sequence of joint positions to the arm.
    b. You build a “goal” message that contains an ordered list of (positions, velocities, time_from_start) for each of the six UR10e joints, send it to the action server, and it drives the arm through that motion profile.

- CommandRobotiqGripperAction

a. This is a custom ROS action (in robotiq_2f_gripper_msgs) for opening/closing the two-finger adaptive gripper.
b. You pack a position (0=open, 255=closed), plus optional speed/force parameters, into a "goal" message and send it to the gripper's action server. It runs the command and then reports back when it's done (or if something went wrong).

In our system there is really a single ROS node,`RobotUR10eGripper`,that hosts both of those action servers. The ChessMovementController simply creates and sends those two kinds of "goals" to that one node, which actually executes the motion on the real arm and gripper.

2.3.2.4 RobotUR10eGripper

- Subscribes to /joint_states and /gripper_joint_states for live feedback.
- Uses a FollowJointTrajectoryAction client on /scaled_pos_joint_traj_controller/follow_joint_trajectory to drive the UR10e's six joints.
- Uses a CommandRobotiqGripperAction client on command_robotiq_action to open and close the gripper.

2.3.2.5 Key Topics & Services

In ROS we communicate two ways:

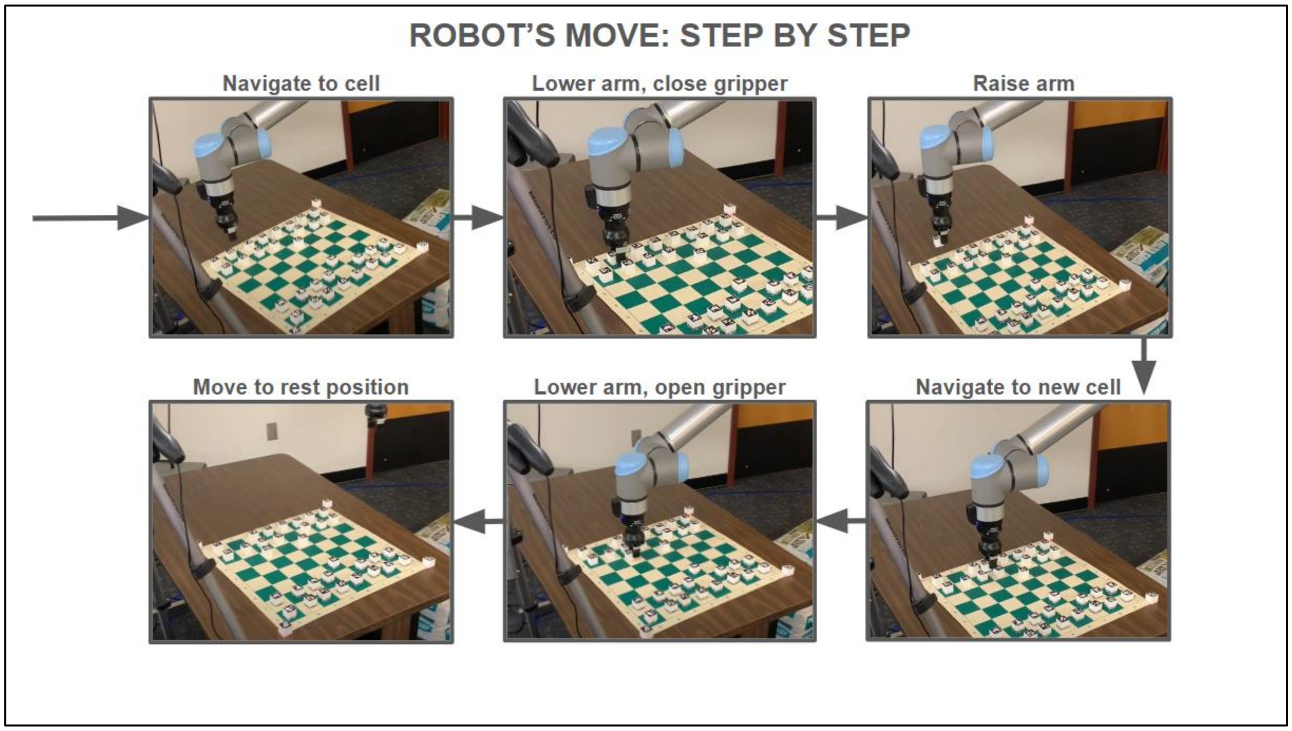
- Topics are like broadcast channels you can publish into or listen to.
  - /joint_states tells us where each of the UR10e's six joints currently is, in real time.
  - /gripper_joint_states does the same for the Robotiq fingers.
  - /scaled_pos_joint_traj_controller/follow_joint_trajectory is the channel where we send a full motion plan – a timed list of joint positions – and the arm controller executes it.
  - command_robotiq_action is an action-style topic where we send an open/close command to the gripper and get back a success/failure result.

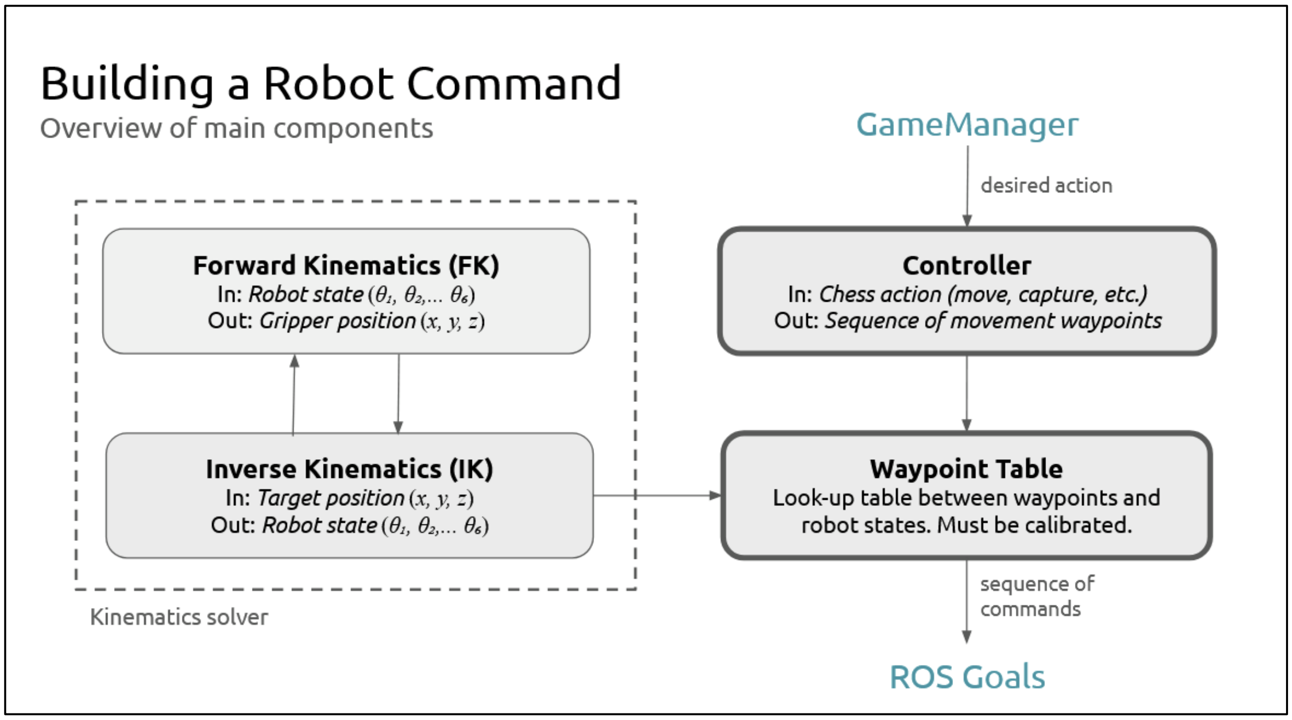
- Services are like remote function calls. Before we can move the arm, we need to make sure the right controller is running.
  - controller_manager/load_controller asks ROS-Control to load a new controller plugin (in our case the scaled position controller).

- controller_manager/switch_controller tells it which controllers to start and which to stop, so we have exactly the interface we need to drive the arm joints.

Together, these topics and services form the plumbing that ties our vision-and-engine code into actual robot motion.

The Robot Control System uses Inverse Kinematics to solve the joint angles required to move from position x to position y. The path planning mechanism is a pick and place mechanism which the arm:




Building a Robot Command
Overview of main components
GameManager
desired action
Forward Kinematics (FK)
In: Robot state ($\theta_1, \theta_2, \ldots \theta_6$)
Out: Gripper position (x, y, z)
Controller
In: Chess action (move, capture, etc.)
Out: Sequence of movement waypoints
Inverse Kinematics (IK)
In: Target position (x, y, z)
Out: Robot state ($\theta_1, \theta_2, \ldots \theta_6$)
Waypoint Table
Look-up table between waypoints and robot states. Must be calibrated.
Kinematics solver
sequence of commands
ROS Goals

2.3.2.6 Forward Kinematics

Forward kinematics is the mathematical foundation that enables our UR10e robotic arm to precisely position its gripper to pick up and place chess pieces on the board. This process maps joint angles to the 3D position of the end effector.

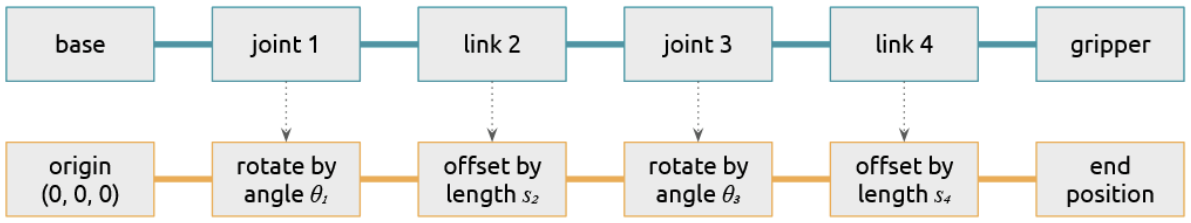
Mathematical Foundation:

Forward kinematics constructs a sequence of transformations that model the physical structure of the robot:

- **Rigid Links:** Physical components that maintain fixed dimensions, represented as translational offsets in 3D space
- **Revolute Joints:** Rotational components that change the orientation between connected links
- **Transformation:** Each link and joint is represented as a 4x4 matrix from Special Euclidean group SE(3), which captures both rotation and translation

Kinematics Chain for UR10e:

As illustrated in the diagram, our implementation models the UR10e as:



- **Base Frame:** The fixed reference joint (origin) from which all measurements begin
- **Joint Transformation:** Each of the 6 controllable joints contributes a rotation by angle θ_i
- **Links Offsets:** Fixed geometric offsets between joints (e.g. upper arm length of 0.613m)
- **End Effector:** The Hand-e gripper positioned at the end of the chain

Computing End Position:

The position of the chess piece gripper is calculated as:

1. Start at the base origin (0,0,0)
2. Apply the first joint rotation (shoulder pan)
3. Add the link offset to the upper arm
4. Apply the second joint rotation (shoulder lift)
5. Continue through all joints and links sequentially
6. Final position = product of all transformation matrices

Implementation in the Chess System:

Our system uses numerical optimization techniques that:

- Compute forward positions for planning safe trajectories between chess squares
- Apply inverse kinematics to determine joint angles needed to reach specific board positions
- Ensure the gripper approaches pieces from above to avoid collisions with other pieces

This mathematical framework enables the precise control needed to smoothly manipulate chess pieces during gameplay, while maintaining safety constraints and avoiding collisions with the physical environment.

2.3.2.7 Inverse Kinematics

Mathematical Foundation of Inverse Kinematics:

Inverse kinematics (IK) solves the critical problem of determining joint configurations that position the robot's end-effector (gripper) at a specific target location. For our UR10e robotic arm, this means finding the six joint angles that place the gripper precisely above a chess piece or square.

Problem definition:

Given a target 3D position **p** on the chessboard, we need to find joint angles **q** such that:

$$\mathrm{IK}(\vec{\mathbf{q}}) = \arg\min_{\vec{\mathbf{q}} \in \Theta} j(\vec{\mathbf{q}})$$

Where our cost function j(q) measures the squared distance between the forward kinematics result and our target:

$$j(\vec{\mathbf{q}}) = \frac{1}{2} \left\| \mathrm{FK}(\vec{\mathbf{q}}) - \vec{\mathbf{p}} \right\|^2$$

Our Implementation Approach:

Unlike simpler robotics problems, chess manipulation requires precise positioning with complex motion constrains:

1. **Numerical Optimization Solution:** We implement IK using the scipy.optimize.least_squares method, which effectively handles our nonlinear search problem with joint constraints.
2. **Partial IK Strategy:** Our approach optimizes only the first three joints (shoulder pan, shoulder lift, elbow) to reach the target position, while mainting a consistent downward-facing orientation needed for chess piece manipulation
3. **Bounded Optimizatio:** We carefully constrain joint angles within mechanical limits:
    a. bone_joints_lower_bounds = [-2π, -π, -ε]
    b. bone_joints_upper_bounds = [2π, ε, π]

   These bounds prevent self-collision and ensure the arm approaches pieces from above.
4. **Convergence control:** Our implementation uses adaptive tolerance parameters and enforced maximum iterations to guarantee reliable solutions even in challengin board positions.

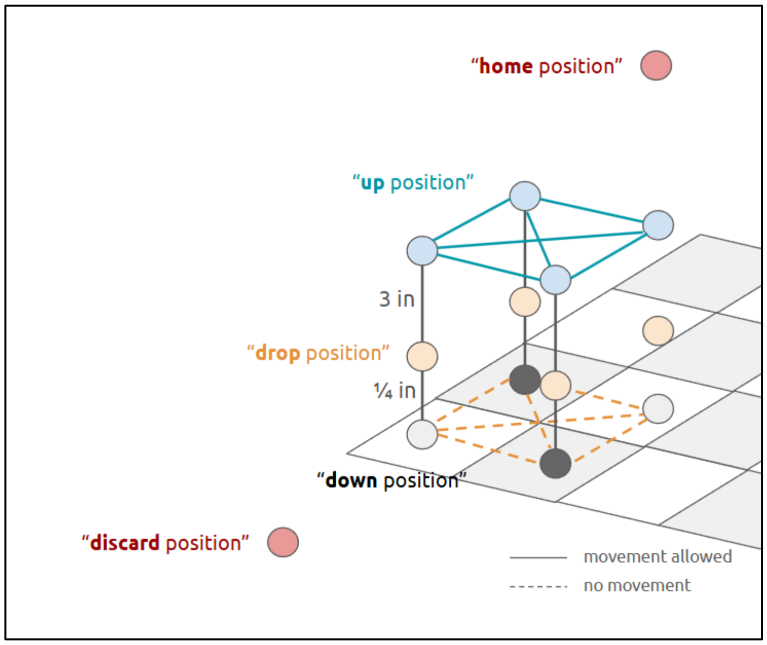
Real-world Application

When executing a chess move, our system:

1. Maps board coordinates (e.g. “e4”) to physical 3D coordinates
2. Solves the IK problem for both pickup and placement positions
3. Generates intermediate waypoints for collision-free paths
4. Executes a smooth trajectory through these points

This mathematical foundation enables our robotic arm to determine precise joint configurations for any reachable position on the chessboard, allowing for accurate chess piece manipulation throghout a game.

2.3.2.8 Waypoints



Waypoints are predefined 3D positions with associated joint configurations that serve as the fundamental building blocks for robot movement planning. In our chess robot system, waypoints represent strategic positions that guide the UR10e arm through the precise sequence required for chess piece manipulation.

The waypoint system consists of:

```
Waypoint
├── label: string identifier (e.g., "a1", "home")
├── pos: 3D vector (x, y, z) in meters
└── jv: JointVector (6 angles for each robot joint)
```

Waypoint Types and Hierarchy

As shown in the diagram, our system uses five distinct types of waypoints:

- Home Position (home) - A safe idle position away from the board
  - Used at the beginning/end of movements
  - Keeps the arm out of the player's way between moves
  - Joint configuration: [2.449, -1.842, 1.006, -2.138, -1.581, 1.389]
- Discard Position (discard and discard_up) - Location for captured pieces
  - Special area beside the board for removed pieces
  - discard_up provides a safe approach path
- Up Position (a1_up, b3_up, etc.) - Hovering position 3 inches above each square
  - Forms safe travel paths between squares
  - Prevents collisions with standing pieces
  - Generated with STANDARD_UP_HEIGHT = 0.08 (meters)
- Down Position (a1, e4, etc.) - Contact position for grabbing pieces
  - Exact position where the gripper meets the chess piece
  - Base level for all 64 squares on the board
- Drop Position (a1_drop, h8_drop, etc.) - Position for releasing pieces
  - Slightly above the board surface (¼ inch)
  - Ensures gentle placement without tipping pieces
  - Calculated with DROP_HEIGHT = 0.005 (meters)

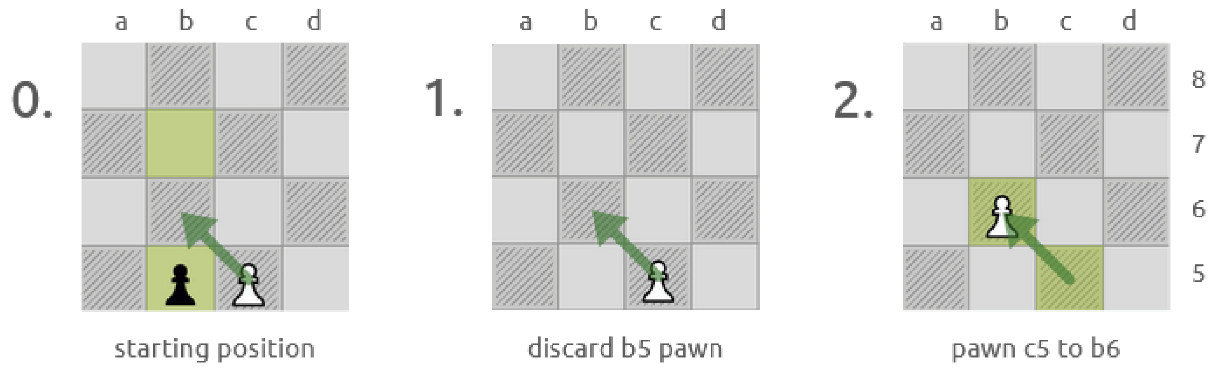
2.3.2.9 Waypoint Calibration

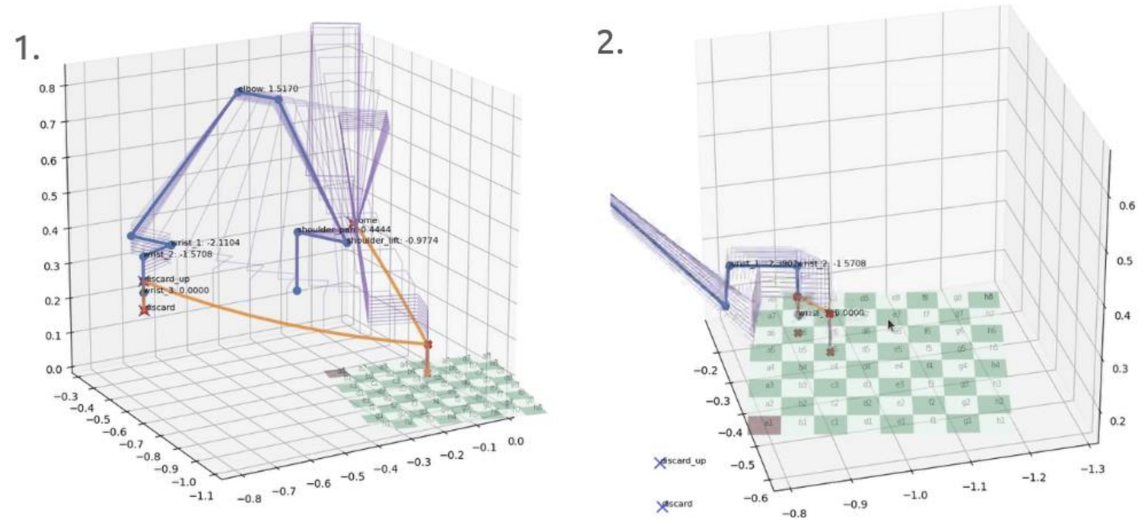
The 194 waypoints (64×3 board positions + 2 special positions) are managed through:

1. Manual Calibration of key reference points:
   a. The a1, h1, a8, and h8 corners define the board's position and orientation
   b. The calibration process records joint angles when the robot is manually positioned at these points
   c. Joint values are captured via the /joint_states ROS topic
2. Algorithmic Generation of all other waypoints:
```
x_hat = (h8 - a8) / 7 # Unit vector along files
y_hat = (h1 - h8) / 7 # Unit vector along ranks
Generate position for any square
pos = a8 + i * x_hat + (8 - rank) * y_hat
```
3. Inverse Kinematics to determine joint angles for each position:
```
For each board position

jv = Skeleton.partial_inverse_kinematics(pos)
```

2.3.2.10 Motion Planning





The pick_and_drop_action_chain function demonstrates how waypoints are used:

1. Move from home to square_up (safe position above source square)
2. Descend to square to grasp the piece
3. Return to square_up with the piece securely held
4. Move to destination_up (safe position above target square)
5. Descend to destination_drop to place the piece
6. Release the gripper and ascend to destination_up
7. Return to home position

2.4 USER CHARACTERISTICS

The user must be of decent physical ability to move the chess pieces and confirm their move.

2.5 CONSTRAINTS

The arm must operate in a way that limits the possibilities of intersecting with an object or person.
The 3 board corners must be calibrated before at the beginning.
The chess pieces must have a good distance between them to allow for the gripper to pass between them while attempting to grab or place a piece.

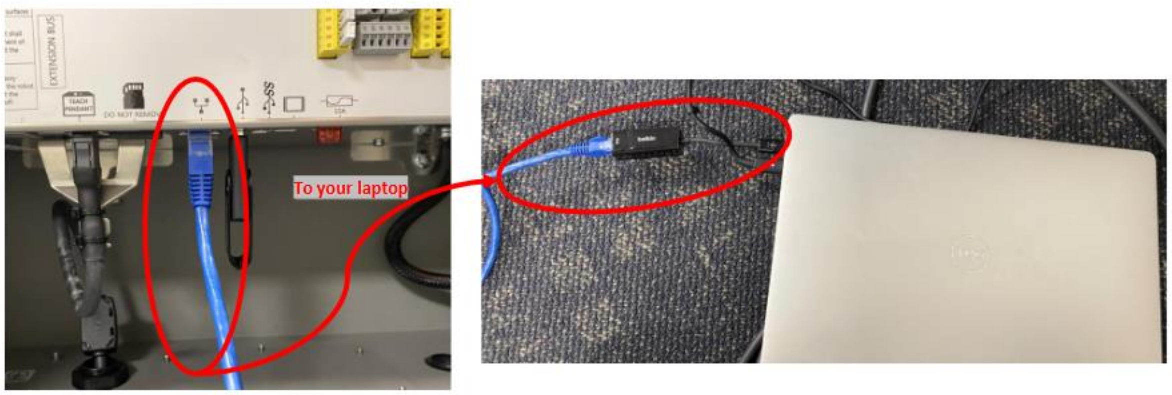
3 Specific Requirements

OS: Ubuntu 20.04
Hardware: UR10e, Hand-e, and a working laptop
Software Interfaces: at least Python 3.8, ROS Noetic
Communications Interfaces: Ethernet cable, and USB-c to ethernet adapter (optional)

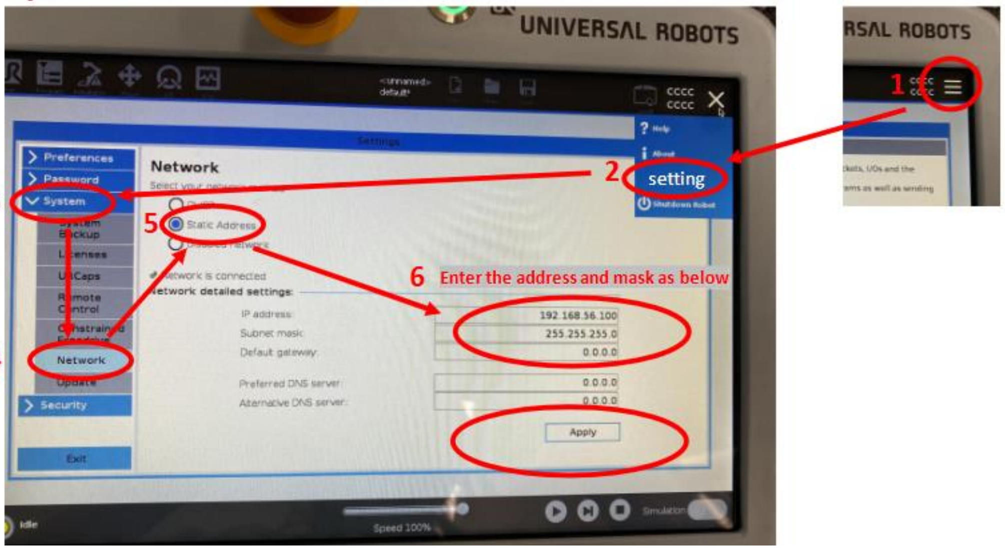
3.1 Hardware:

An individual driver for each of UR10e and Hande is needed. Additionally, the user needs to navigate through the setup of the robot. Below describes how the user should navigate through it (from Dr. Bowen Weng’s presentation slides):

Hardware connection



Setup network on robot



Network
Select your network method

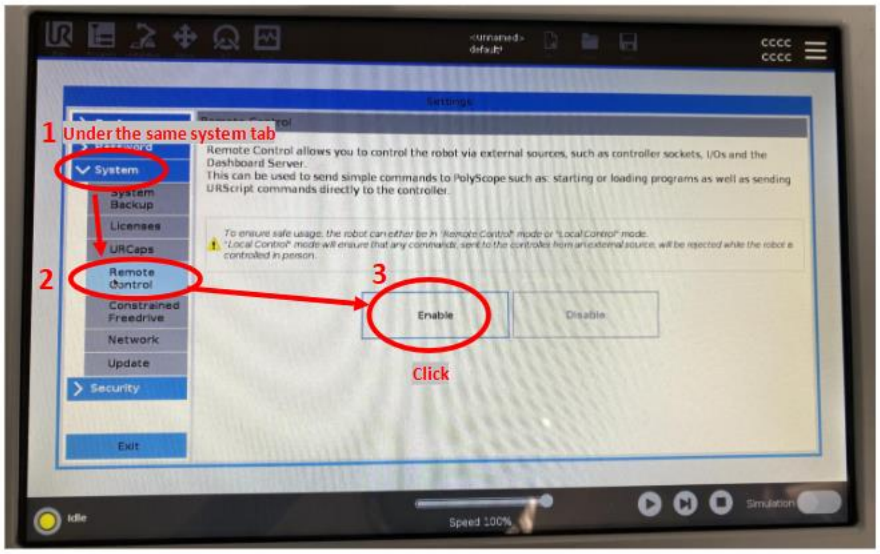
Static Address

Network is connected

Network detailed settings:

IP address:	192.168.56.100
Subnet mask:	255.255.255.0
Default gateway:	0.0.0.0
Preferred DNS server:	0.0.0.0
Alternative DNS server:	0.0.0.0

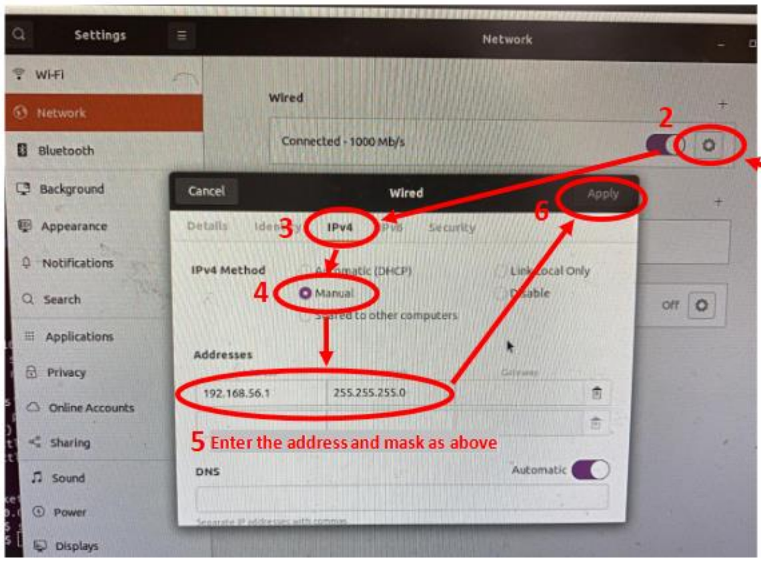
Enable remote control



Remote Control allows you to control the robot via external sources, such as controller sockets, I/Os and the Dashboard Server.
This can be used to send simple commands to PolyScope such as: starting or loading programs as well as sending URScript commands directly to the controller.

To ensure safe usage, the robot can either be in "Remote Control" mode or "Local Control" mode.
"Local Control" mode will ensure that any commands, sent to the controller from an external source, will be rejected while the robot is controlled in person.

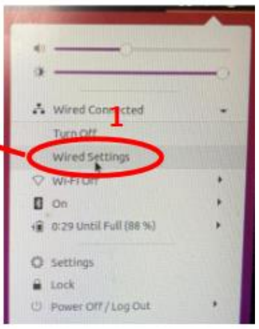
Setup network on laptop



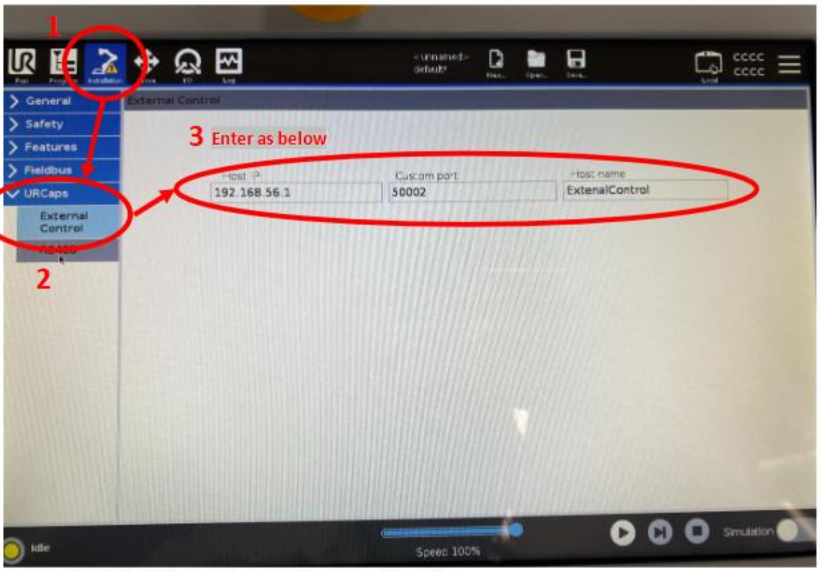
IPv4 Method: Manual

Addresses: 192.168.56.1 255.255.255.0

5 Enter the address and mask as above

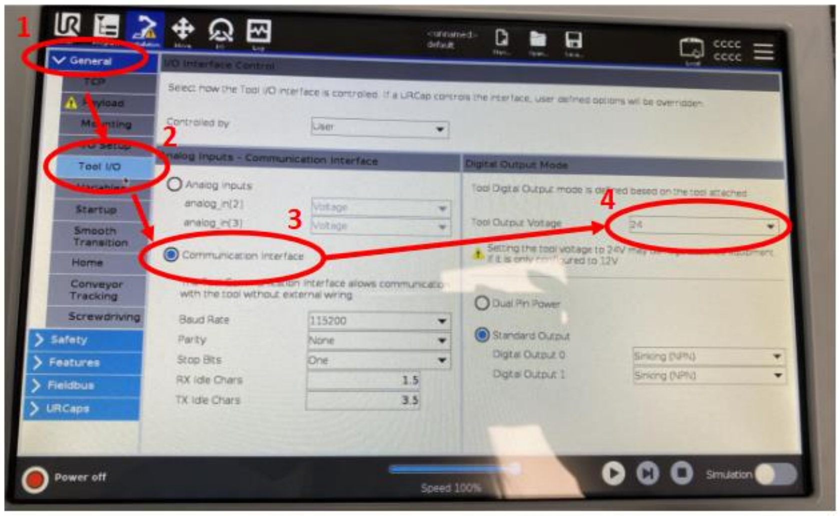


Setup connection to laptop



Host IP: 192.168.56.1
Custom port: 50002
Host name: ExternalControl

Setup tool I/O (if gripper installed)



Select how the Tool I/O interface is controlled. If a URCap controls the interface, user defined options will be overridden.

Controlled by: User

Analog Inputs
analog_in[2]: Voltage
analog_in[3]: Voltage

[✓ Communication Interface]

Communication Interface allows communication with the tool without external wiring

Baud Rate: 115200
Parity: None
Stop Bits: One
RX Idle Chars: 1.5
TX Idle Chars: 3.5

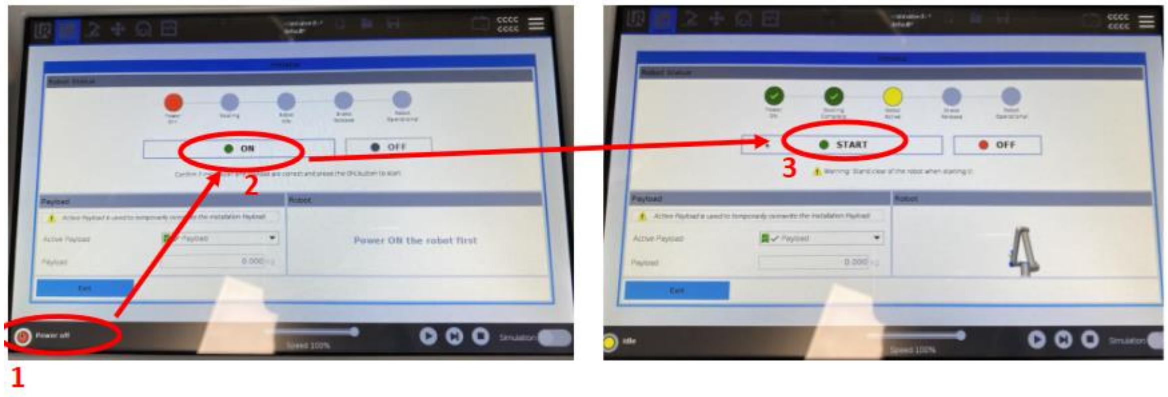
Tool Digital Output mode is defined based on the tool attached

Tool Output Voltage: 24

Setting the tool voltage to 24V may damage equipment if it is only configured to 12V

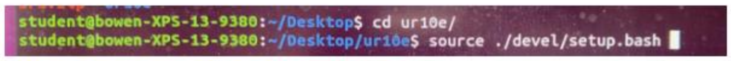
Dual Pin Power
[✓ Standard Output]
Digital Output 0: Sinking (NPN)
Digital Output 1: Sinking (NPN)

Turn on robot



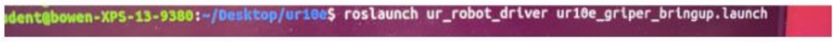
Setup ROS connection

Step1: go to catkin workspace, run: source ./devel/setup.bash
Note: run this every time you open a new terminal



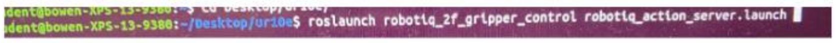
```
student@bowen-XPS-13-9380:~/Desktop$ cd ur10e/
student@bowen-XPS-13-9380:~/Desktop/ur10e$ source ./devel/setup.bash
```

Step2: build connection to the arm
roslaunch ur_robot_driver ur10e_griper_bringup.launch



```
dent@bowen-XPS-13-9380:~/Desktop/ur10e$ roslaunch ur_robot_driver ur10e_griper_bringup.launch
```

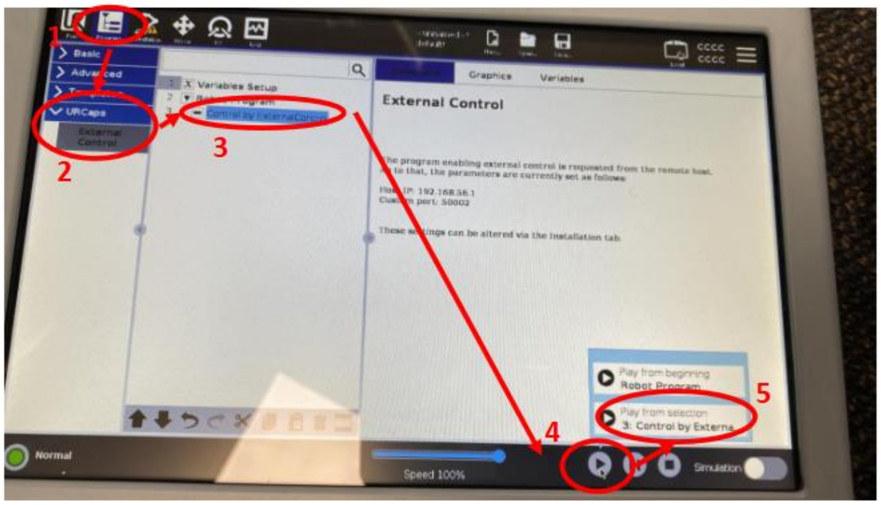
Step3: open a new terminal,build connection to the gripper (remember to source first)
roslaunch robotiq_2f_gripper_control robotiq_action_server.launch



```
dent@bowen-XPS-13-9380:~/Desktop/ur10e$ roslaunch robotiq_2f_gripper_control robotiq_action_server.launch
```

Setup ROS connection

Step4: run external control



3.1 APPENDIX

Here, we provide the necessary information regarding the files that will be helpful to understand how our system works. Other submodules are well documented and are self-explanatory, so we only document the core mechanisms.

Main.py
The main file serves as a starting point for the system.

The following files reside in jh1/core

_game_manager.py
Under the directory, jh1/core, the module _game_manager.py integrates the vision system with the robot control system.

_game_state.py
Handles the gameplay logic and has functions to print out a game state.

_orchestrator.py
Has functions necessary for executing the moves in sequence.

_engine.py
Contains the wrapper class that provides an interface to use the stockfish binary.

ChessMovement.py
It implements the complete robot movement controller responsible for executing chess moves
Handles safety protocols and error recovery
Contains pre-defined positions like "home", "observe", "prepare" and "retreat" that are crucial for safe operation
Implements specific move types (normal moves, captures, castling, en-passant)
Controls gripper operations for piece manipulation
Provides testing functions for board calibration

RobotUR10eGripper
class that directly interfaces with the hardware

Waypoint system
defines movement paths