Project Joseph Hoane: A Robotic Chess Player

FINAL DEMO

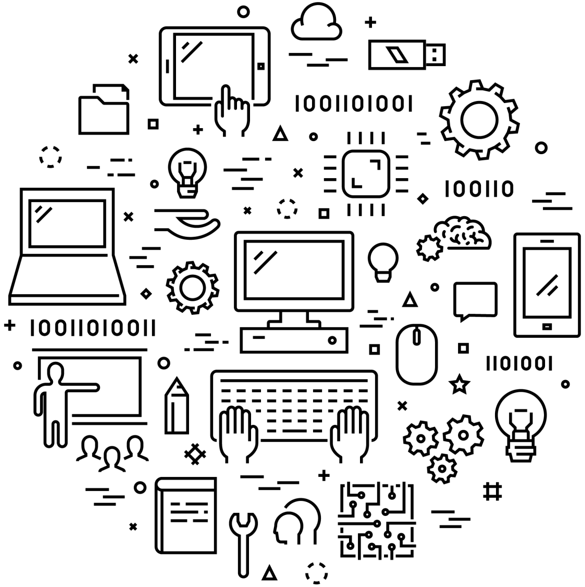
Nhat Anh Bui

Autrin Hakimi

Cal Hokanson-Fuchs

Thanh Mai

Junhyung Shim



Dr. Bowen Weng

Computer Science Department

Iowa State University

Team SD02

05/13/2025

Overall Recap

Demo 1: Initial Research

- Chess engine
- Research: arm simulation
- Research: apriltags for corner detection

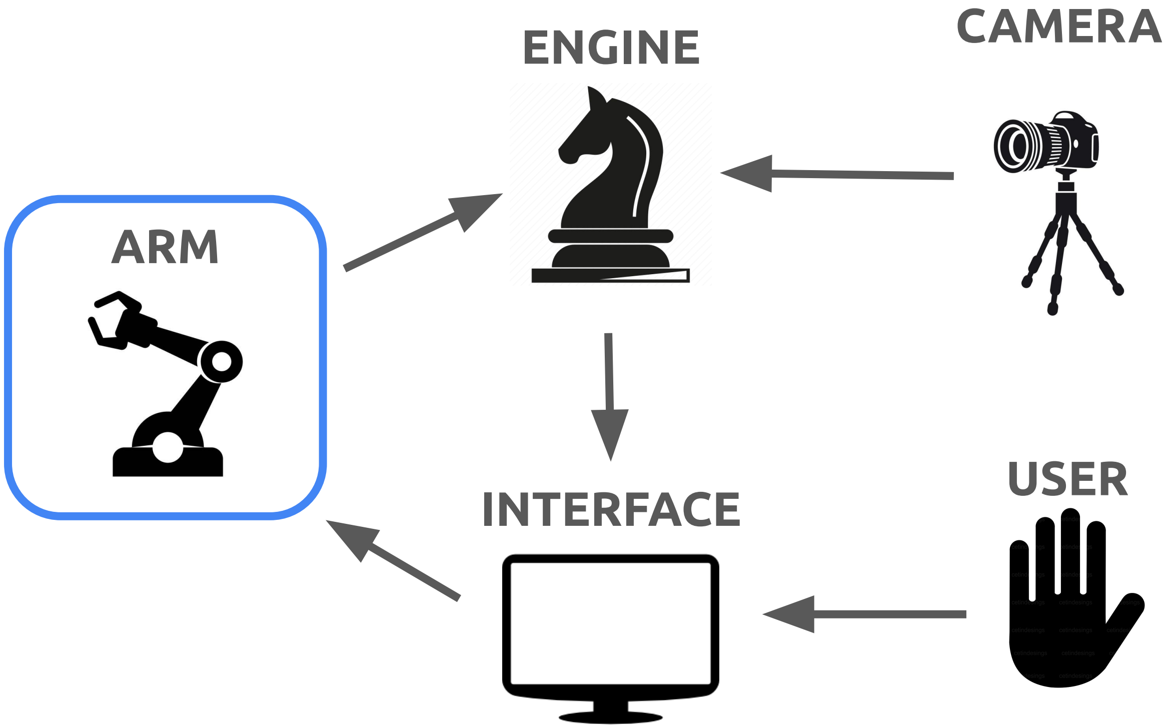
Demo 2: Detection

- Full board detection with apriltags
- Gameplay with piece movement
- User Interface

Demo 3: Robotics

- Integrate robotics with board detection and chess modules
- Use driver codes (work with hardware)
- Live arm movement using inverse kinematics

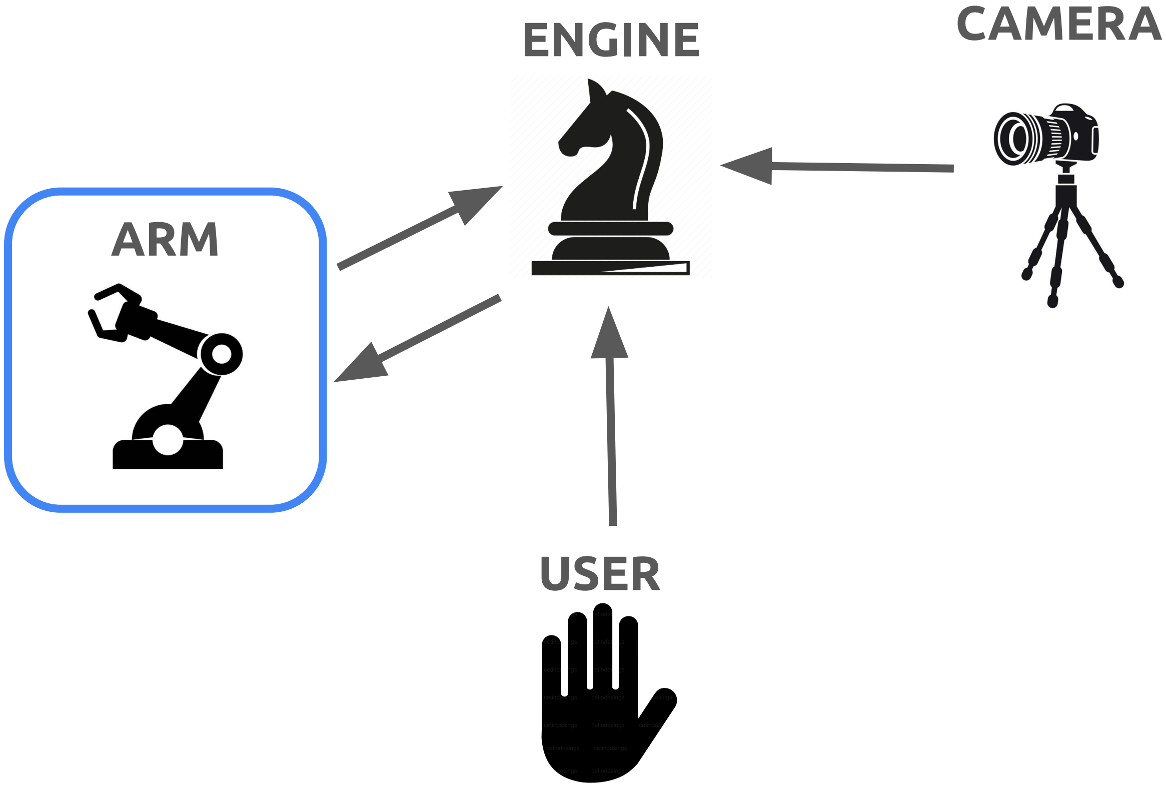
Intended Game Loop



Successes

- Camera, Chess Engine, and Robotic Arm modules have been connected
- Robot is very accurate and safe in its operation
- Detection is robust and accurate under various lighting conditions
- Chess Engine usage is bug-free and quick in its calculations

Actual Game Loop



Board Detection

Board detection relies on computer vision using AprilTags

- **We used AprilTags because**
  - Fast
  - Robust to lighting conditions
  - Less dependencies

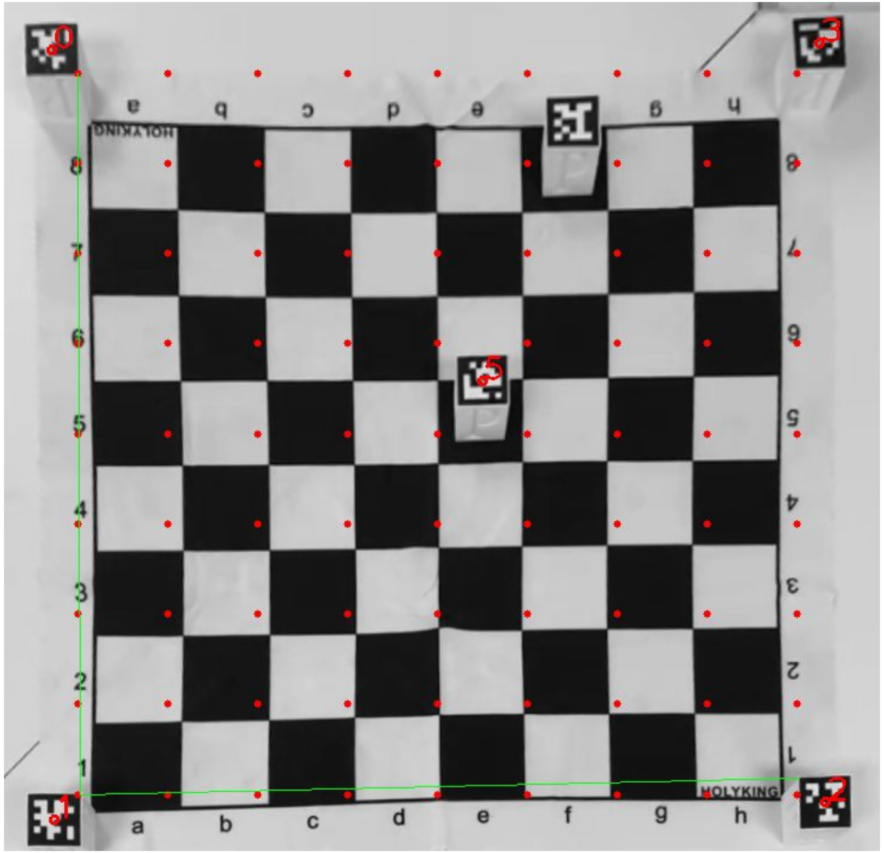
- **ML/DNN Techniques can be employed but…**
  - Training/fine-tuning time overheads
  - Trade-off between robustness and performance
  - Less predictable
  - Harder to debug and test

Image sources: https://www.ucf.edu/news/ucf-researchers-develop-technology-for-ai-that-mimics-the-human-eye/
https://april.eecs.umich.edu/software/apriltag

Board Detection

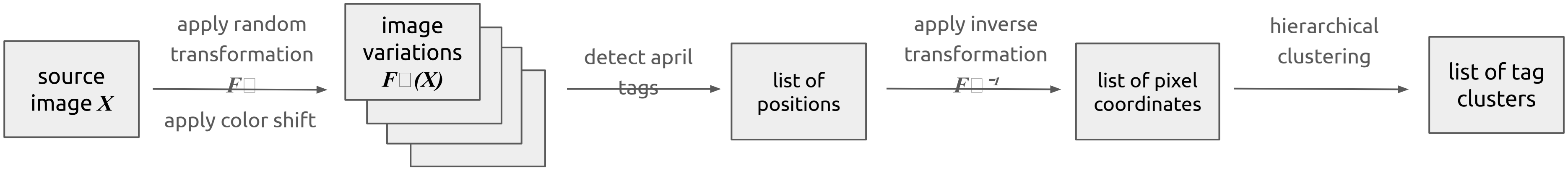
Problem:

- Not all tags were getting detected
- Many factors including:
    - Tags can be too small
    - Chess board pattern confusing the detector
    - Paper sticker prone to glare

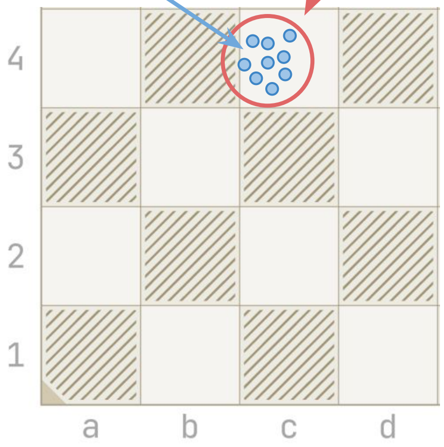


Solution: “Perturb-Detect-Cluster”

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



From Tags to Pieces

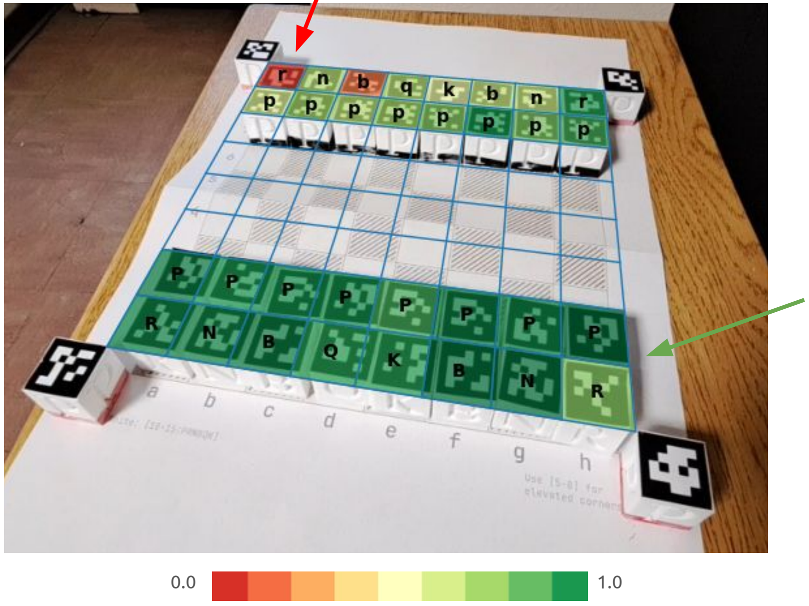


individual detections

overall cluster

for each cluster, snap to a square on the board

harder tag (far, skewed)



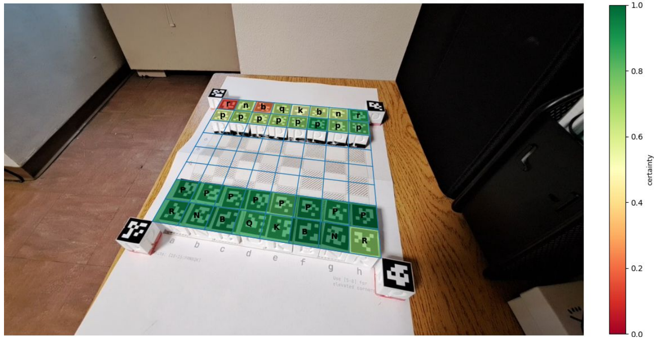
easier tag (near, flat)

$$\text{certainty} = \frac{\#\text{actual detections}}{\#\text{copies made}}$$

Board Detection

Possible Improvements:

- Statistics is run by a single thread, and is applied one after another
- But each transformation is independent of each other
- We can possibly distribute the workload using Python/C interfaces (e.g. OpenMP)



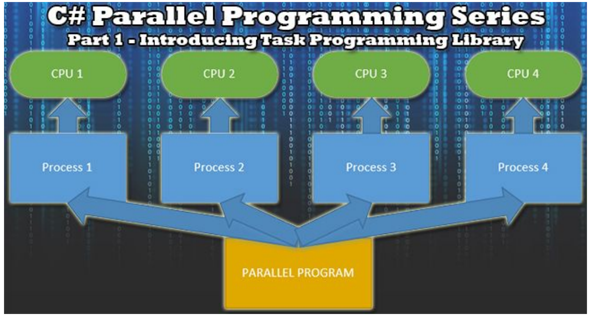
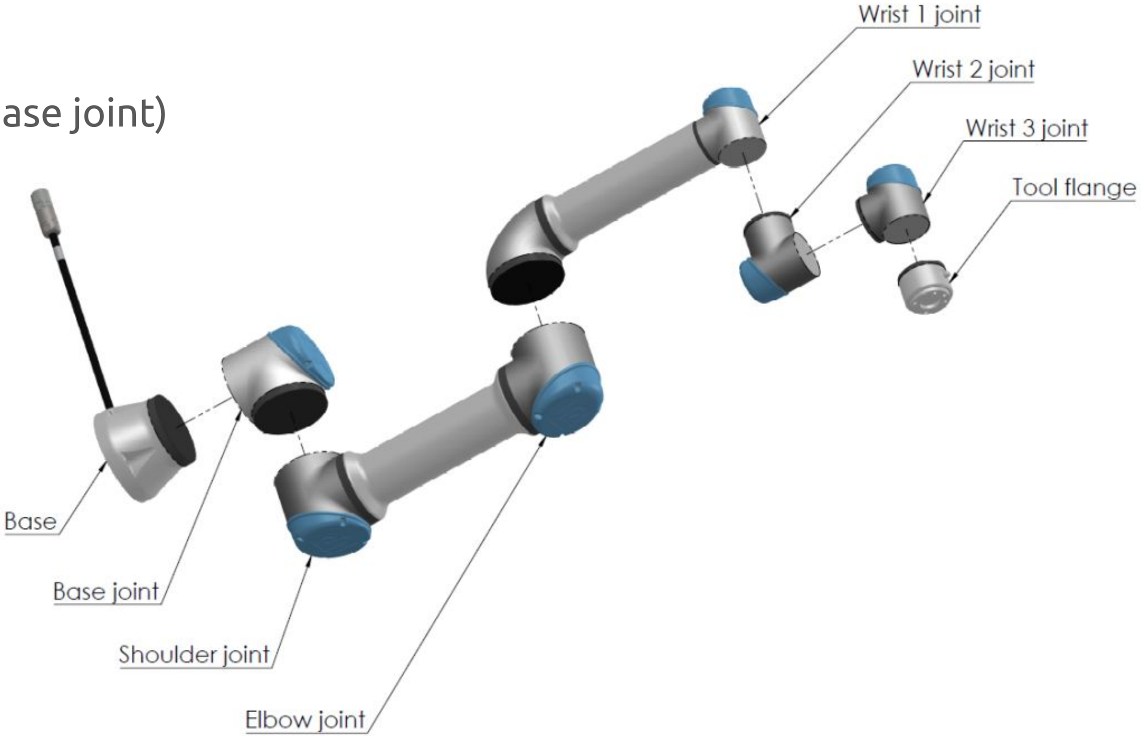

C# Parallel Programming Series
Part 1 - Introducing Task Programming Library
CPU 1
CPU 2
CPU 3
CPU 4
Process 1
Process 2
Process 3
Process 4
PARALLEL PROGRAM

Image Source: *https://www.c-sharpcorner.com/UploadFile/f9f215/parallel-programming-part-1-introducing-task-programming-l/*

The Robot Hardware

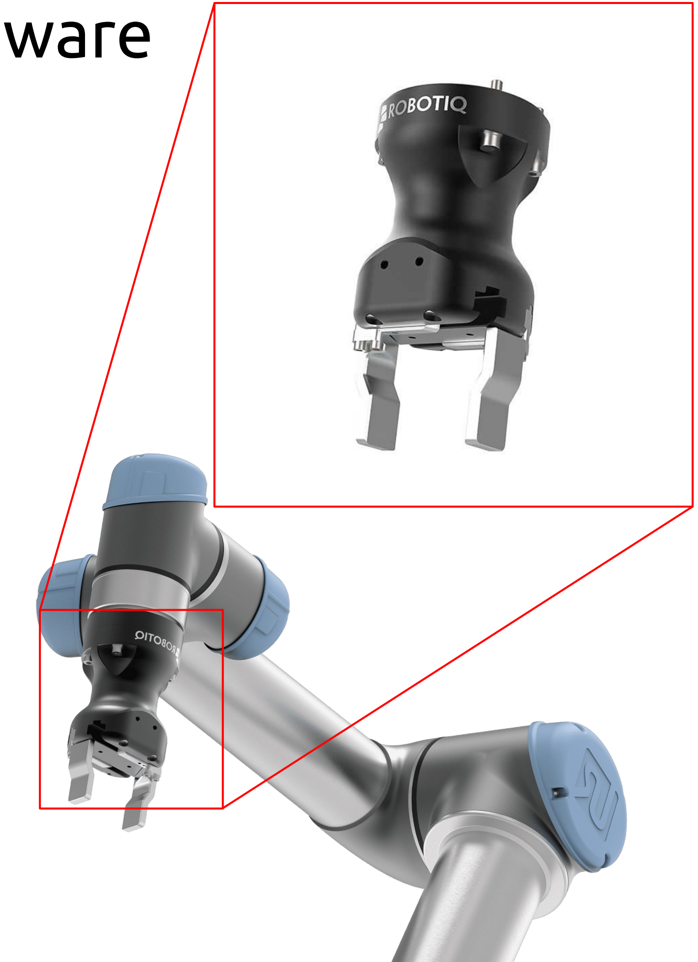
Arm: Universal Robots **UR10e**

- 6 degrees of freedom:
  - `shoulder_pan_joint` (base joint)
  - `shoulder_lift_joint`
  - `elbow_joint`
  - `wrist_1_joint`
  - `wrist_2_joint`
  - `wrist_3_joint`



The Robot Hardware

- **Gripper**: Robotiq **Hand-E** Adaptive Gripper
  - Two adaptive fingers with 2-inch span
  - Position-controlled: Command open/close and read back joint state
  - Infer a successful grasp by monitoring the reported position



System Architecture & Communication
ROS nodes & data flow:

- `GameManager`
  1. Grabs camera frames → AprilTag detections → constructs board state (FEN)
  2. Invokes Stockfish engine → returns best move in UCI format

- `ChessMovementController`
  1. Gets the UCI strings (e.g. “`e2e4`”).
  2. Converts each move into a tiny “pick-and-place” motion plan:
  3. Pre-grasp above source square
  4. Descend, close gripper, lift
  5. Transit above destination
  6. Descend, open gripper, retract
  7. Packages those waypoints into ROS trajectory goals (`FollowJointTrajectory` and `CommandRobotiqGripperAction`)

FollowJointTrajectory & gripper ActionGoals

Two ROS-action interfaces used to talk to the hardware

- FollowJointTrajectoryAction
  - ROS action for issuing a sequence of joint positions to arm
  - Build a "goal" message: an ordered list of (positions, velocities, time_from_start) for each of the six UR10e joints → send it to action server → drives arm through that motion profile → reports back when done
- CommandRobotiqGripperAction
  - Custom ROS action for opening/closing the gripper
  - Pack a position (open/closed) and optional speed/force parameters → "goal" message → send it to the gripper's action server → runs the command → reports back when done

FollowJointTrajectory & gripper ActionGoals

Only a single ROS node, RobotUR10eGripper → hosts both of those action servers

creates & sends

The ChessMovementController ——------> those two "goals" to that one node → execute the motion

System Architecture & Communication
ROS nodes & data flow:

- `RobotUR10eGripper` (ROS node)
  - Subscribes to `/joint_states` and `/gripper_joint_states` for real-time feedback
  - Arm: uses a `FollowJointTrajectoryAction` client to drive the UR10e's 6 joints
  - Gripper: uses a `CommandRobotiqGripperAction` client to open/close the adaptive gripper

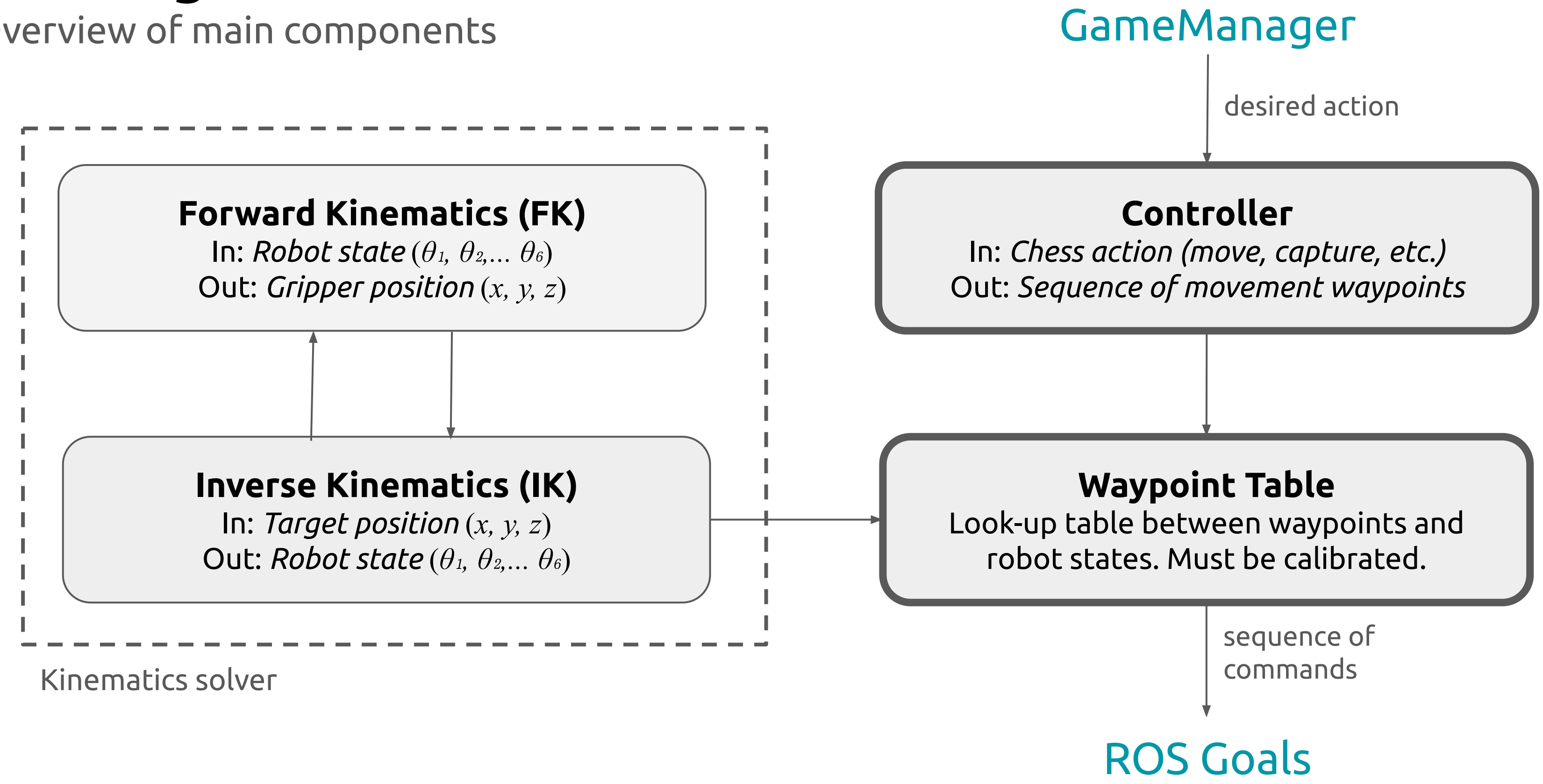
Key Topics & Services

- Topics (broadcast channels):
  - `/joint_states` (robot arm positions)
  - `/gripper_joint_states` (gripper feedback)
  - `/scaled_pos_joint_traj_controller/follow_joint_trajectory` (arm commands)
  - `command_robotiq_action` (gripper commands)

- Controller Manager Services (startup & switching) (to ensure the right controller is running)
  - `controller_manager/load_controller`
  - `controller_manager/switch_controller`

Building a Robot Command

Overview of main components



GameManager

desired action

Controller
In: *Chess action (move, capture, etc.)*
Out: *Sequence of movement waypoints*

Forward Kinematics (FK)
In: *Robot state* $(\theta_1, \theta_2, \ldots \theta_6)$
Out: *Gripper position* (x, y, z)

Inverse Kinematics (IK)
In: *Target position* (x, y, z)
Out: *Robot state* $(\theta_1, \theta_2, \ldots \theta_6)$

Kinematics solver

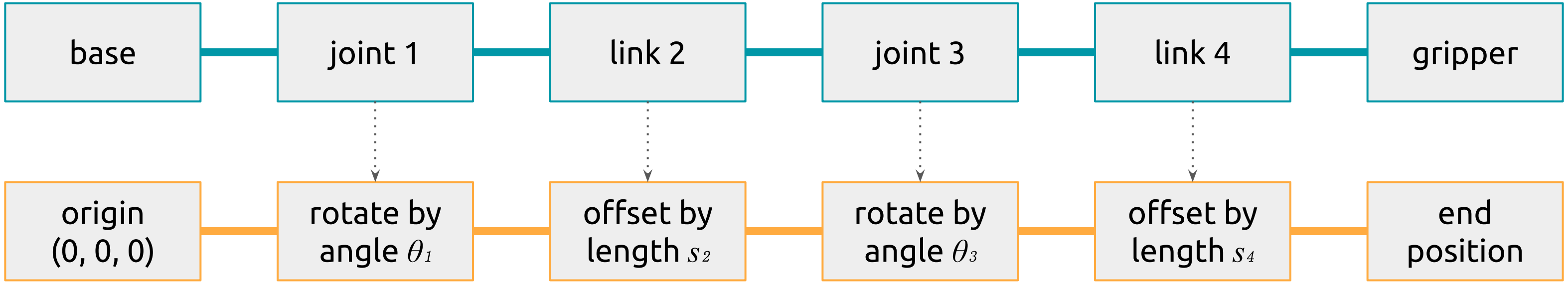
Waypoint Table
Look-up table between waypoints and robot states. Must be calibrated.

sequence of commands

ROS Goals

Forward Kinematics

- Treat each physical part of the robot as a *transformation*:
  - Links are positional offset
  - Joints are rotations
- All of these transformations belongs to the same mathematical family
  - Special Euclidean group $SE(3)$ of **rigid motions**, elements are 4x4 homogeneous matrices
- Net effect: brings origin to end position
  - Overall transformation = product of every individual transformation matrix



Inverse Kinematics

- Given a target 3D position, find the joint angles such that the gripper ends up at the target.
- Nonlinear problem due to rotational (revolute) joints
- Simplest approach: use a nonlinear optimizer from libraries (e.g. `scipy`)
  - Minimize cost function: the squared error from target $\boldsymbol{p}$

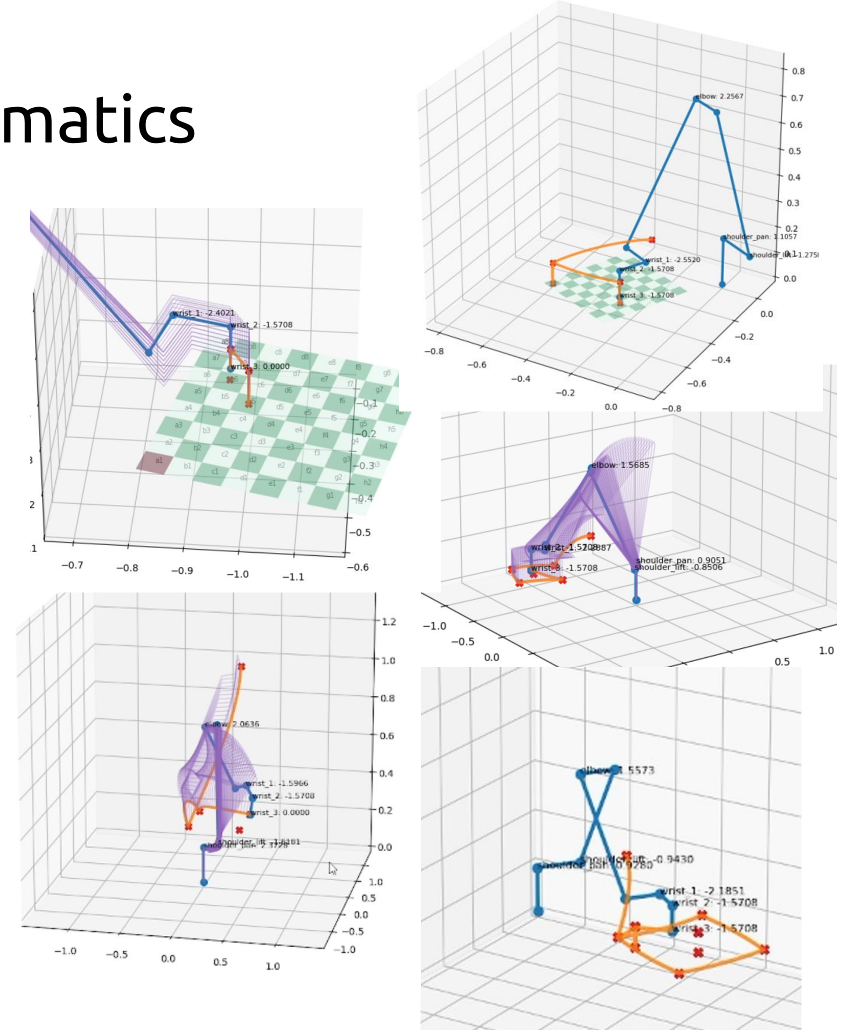
$$j(\vec{\mathbf{q}}) = \frac{1}{2}\left|\left|\mathrm{FK}(\vec{\mathbf{q}}) - \vec{\mathbf{p}}\right|\right|^2$$

  - Where $\boldsymbol{q}$ is a tuple of 6 angles, in parameter space $\boldsymbol{\Theta} \subseteq \boldsymbol{T}^6$ with constraints and known bounds to prevent unsafe configurations:

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

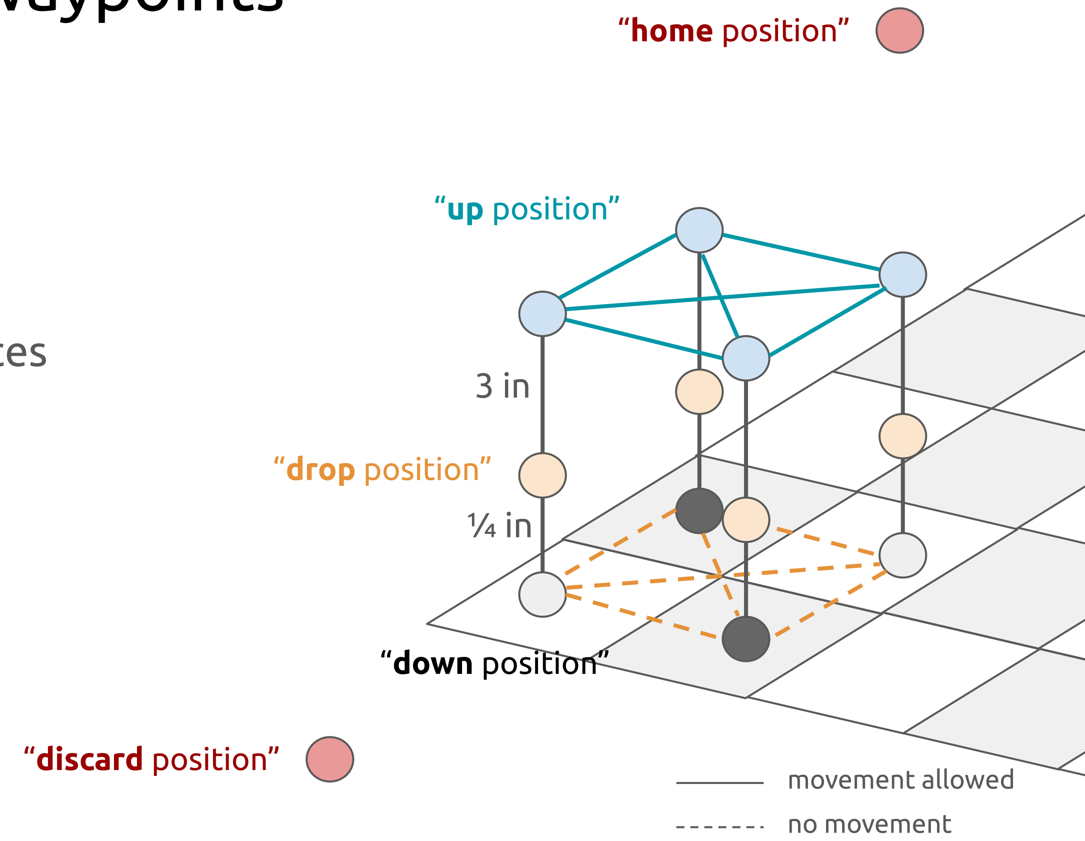
Inverse Kinematics

- In practice, only first 3 joints out of 6 need to be solved
    - Remaining 3 joints used to ensure gripper is level and aligns with the board
- **Result:** can plan arbitrary paths through space, using position sequences



Waypoints

- Final component, maps physical positions to robot states
- ***Home*** position for idling
- ***Discard*** position for captured pieces
- 3 waypoints for each square:
  - ***Up***: directly above
  - ***Down***: pick up pieces
  - ***Drop***: drop off pieces
- Total: 64x3 board + 2 special waypoints

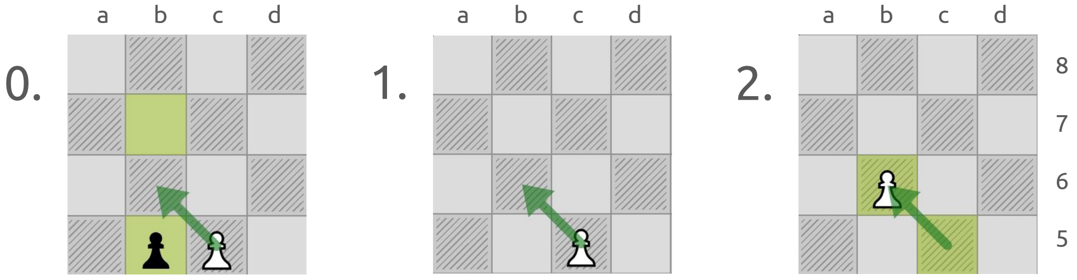


Waypoint Calibration

- Calibration: Manually moving the arm to a position, then recording the angles
  - Angles are read from the `/joint_states` topic
- Home and discard positions must be calibrated separately
- All 64x3 board waypoints can be generated from calibration of 3 corners

Motion Planning

Example: En passant

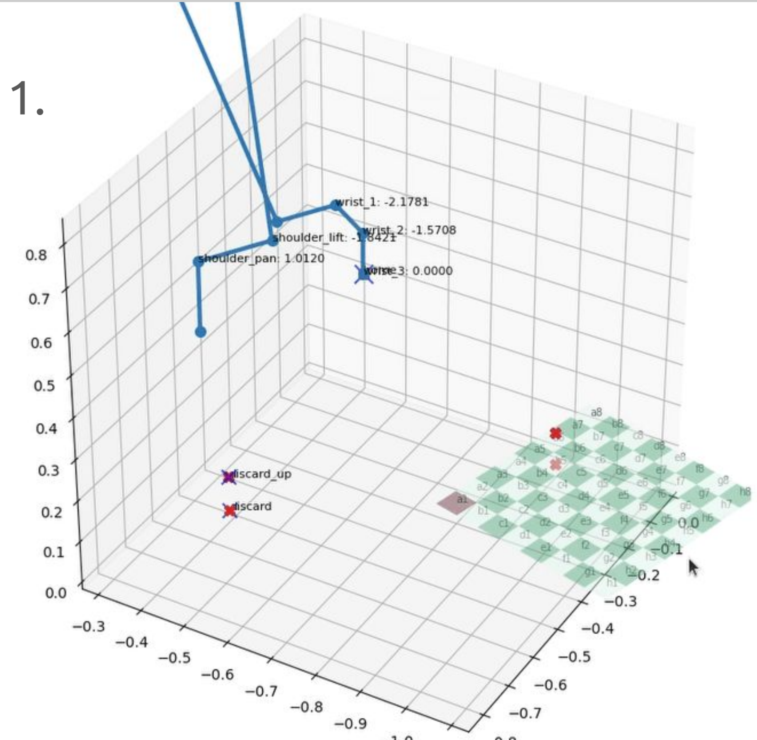


starting position

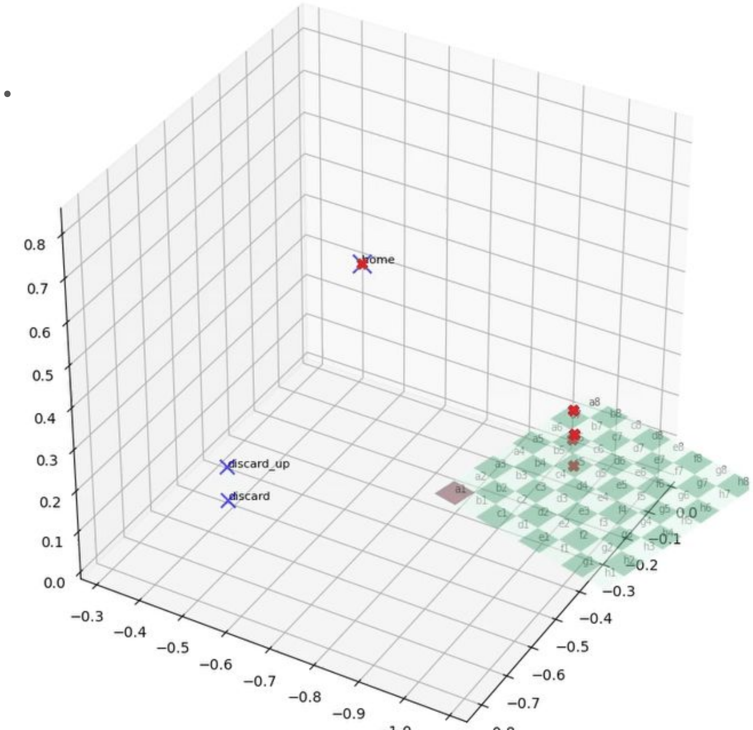
discard b5 pawn

pawn c5 to b6

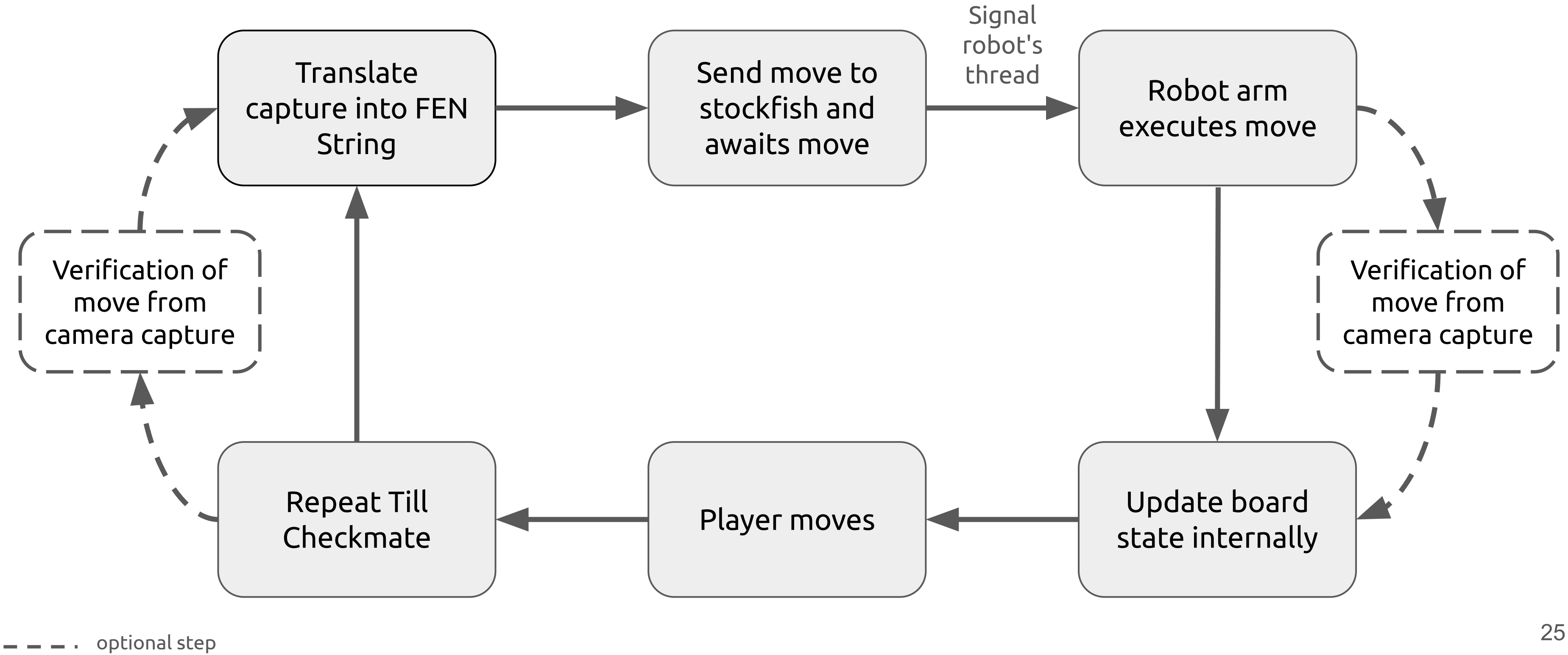
1.



2.



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
Update board state internally
Player moves
Repeat Till Checkmate
Verification of move from camera capture
optional step

Demo Video

Challenges

- Lack of enough lab hours for robot testing
- Lack of support for the gripper component
- Gripper goes offline intermittently
- Unnatural gripper shape
- No feedback sensors in gripper
- UI was unable to be integrated due to time constraints

vs

Acknowledgement

Thank you, Dr. Mitra for the opportunity to do the project.

Thank you, Dr. Bowen and Ling Tang for their guidance with programming the robot arm.