CS4910: Deep Learning for Robotics

David Klee
klee.d@northeastern.edu

T/F, 3:25-5:05pm
Behrakis Room 204

https://www.ccs.neu.edu/home/dmklee/cs4910_s22/index.html

https://piazza.com/northeastern/spring2022/cs4910a/home

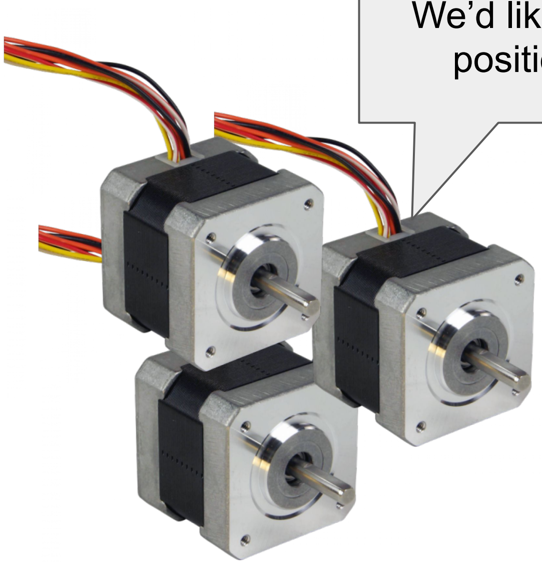
Motion Planning

Today’s Agenda

1. Learn how to move robotic arm
2. Grab a drink *(in pybullet)*
3. Talk about robotic manipulation setups
4. Install Pytorch
5. Solve XOR

How to represent motions

We'd like joint positions



Robot servo actuators

Cartesian coordinates make more sense to me

Human programmer

Sending motor commands in Pybullet

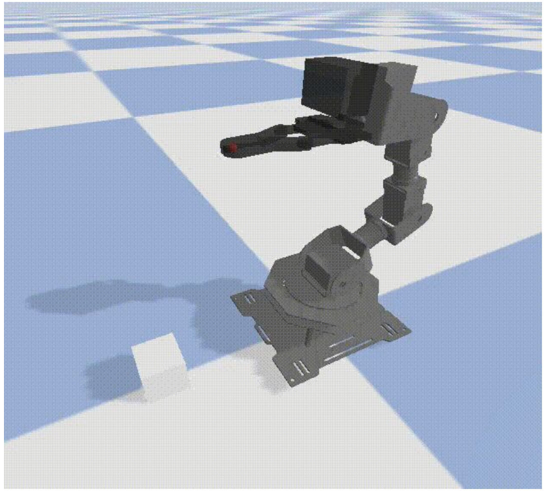
```
pb.setJointMotorControlArray(bodyUniqueId: int,
                             jointIndices: List[int],
                             controlMode: int=pb.POSITION_CONTROL,
                             targetPositions: List[float],
                             targetVelocities: List[float],
                             forces: Optional[ List[float] ],  # max force
                             positionGains: Optional[ List[float] ],
                             velocityGains: Optional[ List[float] ])
```

See Quickstart Guide for information on other controlModes

It is important to set the gains correctly when manipulating objects

Simplest option is to limit velocity:

```
for ji, jp in zip(arm_joint_ids, arm_jpos):
    pb.setJointMotorControl2(robot_id,
                             ji,
                             pb.POSITION_CONTROL,
                             jp,
                             positionGain=0.1,
                             maxVelocity=0.8)
```

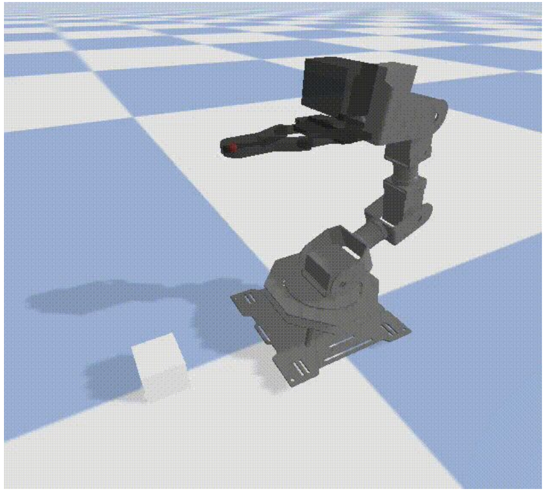


Poorly tuned arm movement results in unstable object manipulation

It is important to set the gains correctly when manipulating objects

Other options for improved grasp stability:

- Increase friction values of object
- Reduce weight of object
- Increase gripper force



Poorly tuned arm movement results in unstable object manipulation

Pybullet joints are modeled as motors, so they will ‘hold’ their position by default

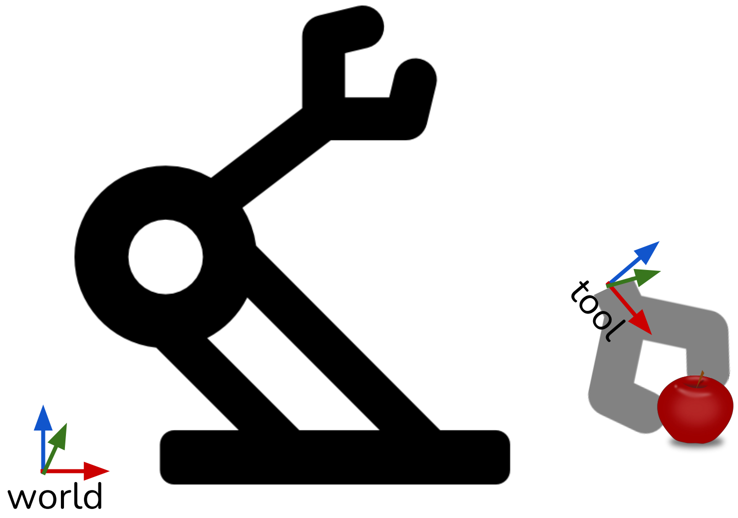
If you want a joint to move freely or to simulate a non-active motor…

```
pb.setJointMotorControl2(robot_id,
                         joint_id,
                         pb.POSITION_CONTROL,  #or pb.VELOCITY_CONTROL
                         force=0)
```

or set force to a small number to simulate joint friction.

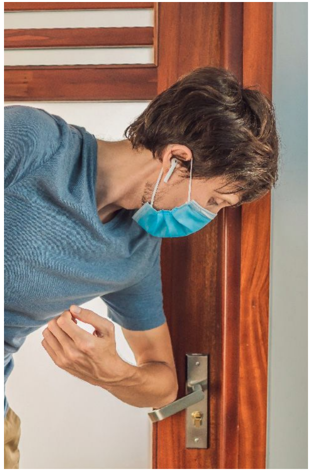
You can use pb.setJointMotorControlArray too

For many robotic manipulation tasks, we care about tool pose.
Predicting actions according to tool pose can simplify learning.

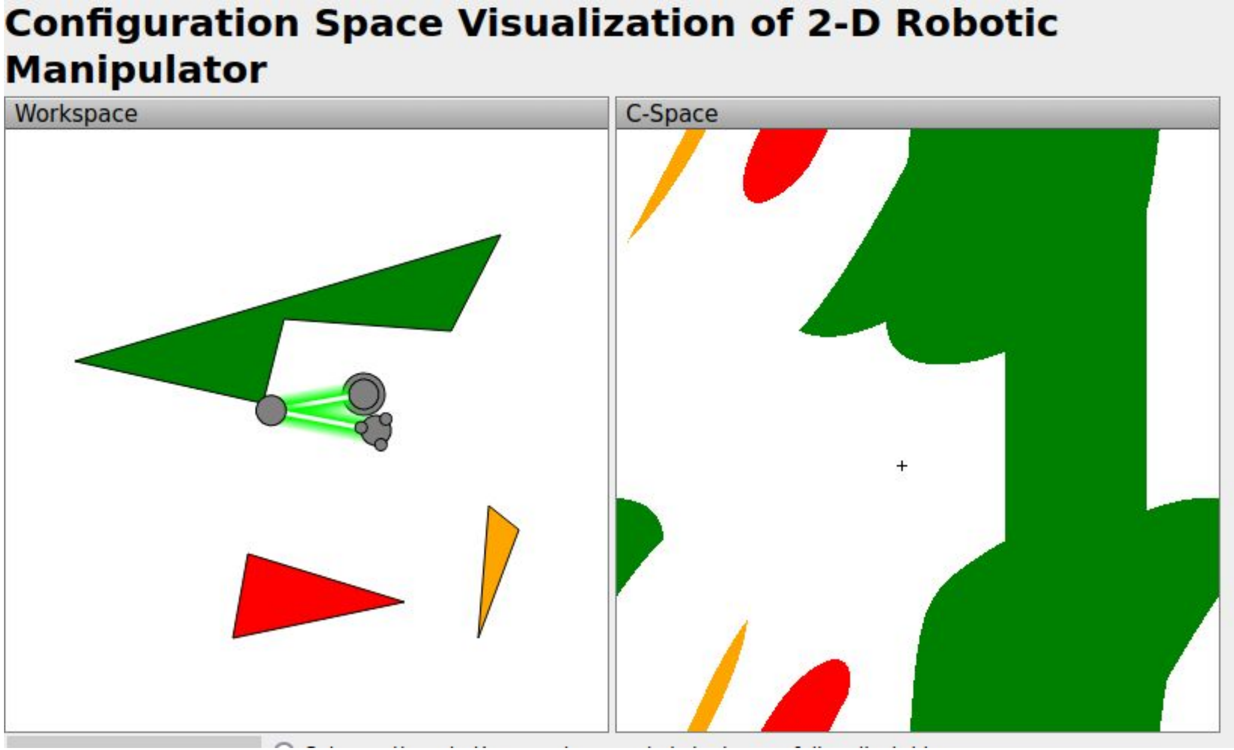


tool == end effector == gripper link

A tangent about joint space (i.e. configuration space)



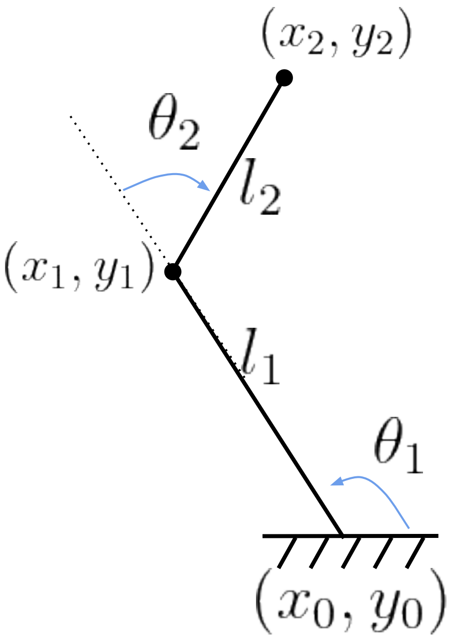
A tangent about joint space (i.e. configuration space)



By projecting obstacles to C-space, planning trajectories becomes a shortest paths problem in a N-dimensional space

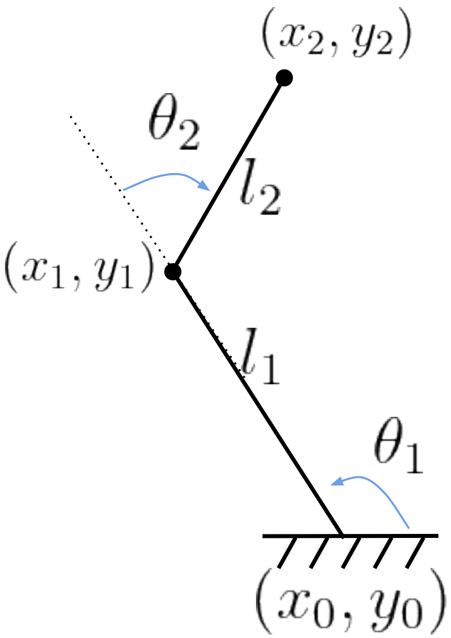
Interactive C-space for 2-link arm: https://www.cs.unc.edu/~jeffi/c-space/robot.xhtml

Forward Kinematics: Calculating tool pose from joint positions



Find (x_2, y_2) in terms of (x_0, y_0)

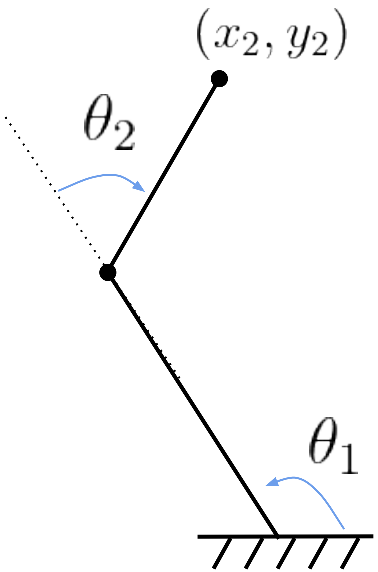
Forward Kinematics: Calculating tool pose from joint positions



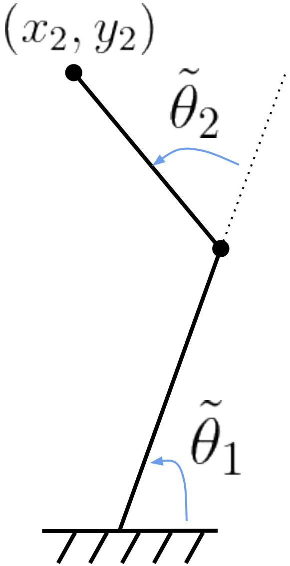
$$x_1 = x_0 + l_1 \cos(\theta_1)$$
$$y_1 = y_0 + l_1 \sin(\theta_1)$$
$$x_2 = x_1 + l_2 \cos(\theta_1 - \theta_2)$$
$$y_2 = y_1 + l_2 \sin(\theta_1 - \theta_2)$$

$$x_2 = x_0 + l_1 \cos(\theta_1) + l_2 \cos(\theta_1 - \theta_2)$$
$$y_2 = y_0 + l_1 \sin(\theta_1) + l_2 \sin(\theta_1 - \theta_2)$$

Inverse Kinematics: calculating joint positions for tool pose



Find θ_1, θ_2 given (x_2, y_2)



There are many joint angles that result in the same end-effector position, thus solving inverse kinematics is not as simple as forward kinematics

Inverse Kinematics as an optimization problem

jacobian matrix

end-effector velocity → $\dot{x} = \mathbf{J}\dot{q}$ ← joint velocities

Good resource on jacobians for robotics

Inverse Kinematics as an optimization problem

$$
\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} = \begin{bmatrix} \frac{\delta x}{q_1} & \frac{\delta x}{q_2} & \cdots & \frac{\delta x}{q_n} \\ \frac{\delta y}{q_1} & \frac{\delta y}{q_2} & \cdots & \frac{\delta y}{q_n} \\ \vdots & \vdots & \vdots\vdots\vdots & \vdots \\ \frac{\delta \gamma}{q_1} & \frac{\delta \gamma}{q_2} & \cdots & \frac{\delta \gamma}{q_n} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix}
$$

We can calculate Jacobian by differentiating forward kinematics equations w.r.t. each joint position

Good resource on jacobians for robotics

Inverse Kinematics as an optimization problem

$$\text{Given } q, x_g, \alpha :$$

$$
\begin{aligned}
x &= \texttt{forward\_kinematics}(q) \\
\Delta x &= \alpha(x_g - x) \\
\mathbf{J} &= \texttt{compute\_jacobian}(x) \\
\Delta q &= \mathbf{J}^{-1}\Delta x \\
q &= q + \Delta q
\end{aligned}
$$

Note that the final joint positions are dependent on the initial joint positions

Pseudo-inverse can be used on Jacobian, which allows for non-square matrices

[Good resource on jacobians for robotics]

Performing Inverse Kinematics in Pybullet

```
pb.calculateInverseKinematics(
    bodyUniqueId: int,
    endEffectorLinkIndex: int,              # use pb.getJointInfo to find this id
    targetPosition: vec3,
    targetOrientation: Optional[vec4], # orientation ignored if omitted or None
    jointDamping: List[float]               # same length as number of joints
    maxNumIterations: int,
    ...,
) -> List[float]
```

Note: output will be list of *all* joint positions. For arm with gripper, you should ignore the values for the gripper joints

Tricks for best experience using end-effector control

1. Start IK from a nearby joint configuration (do not have a singularity!)
   - Use `pb.getJointStates` and `pb.resetJointState` to instantly *teleport* arm

2. Ensure that your target pose is actually reachable (xArm is underactuated!)
   - Precompute workspace in advance

3. Check residuals before proceeding with motor command

Demo: grabbing a drink

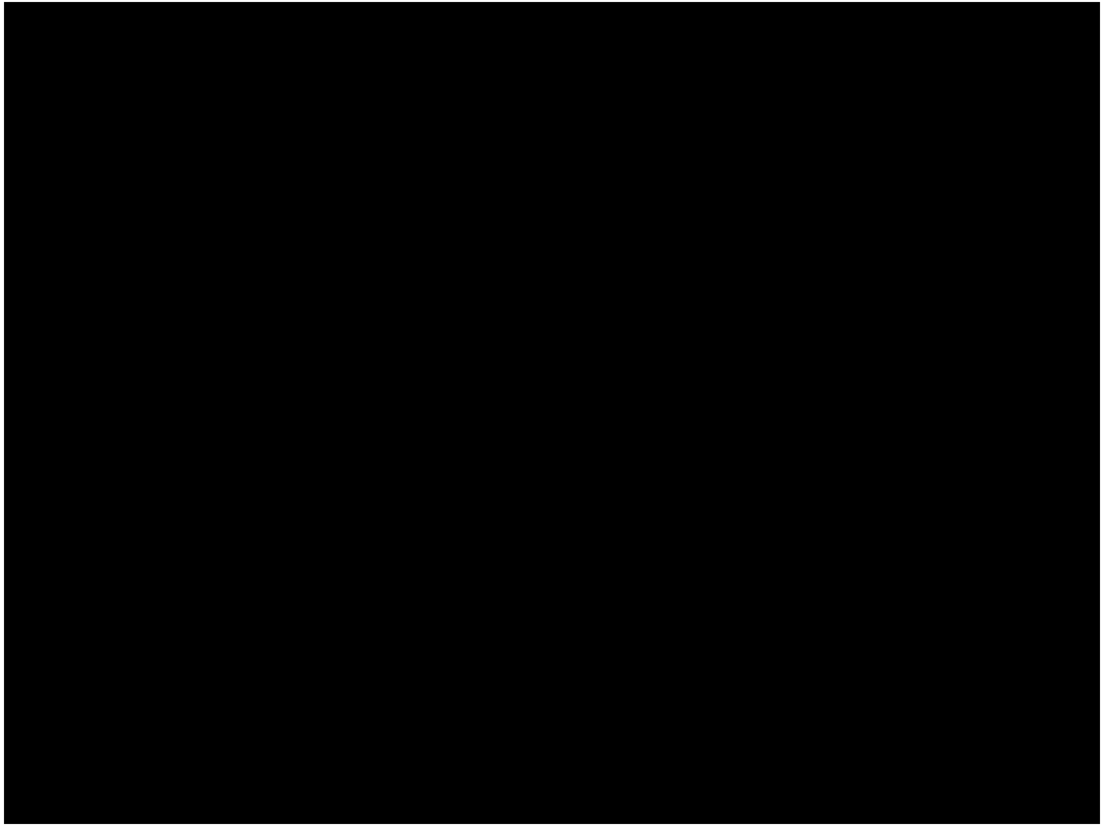
Have it start from singularity to show issue

Talk about dummy end effector index to represent grasping location instead of wrist

Mention iterative process

Performing complex motions with inverse kinematics

Performing complex motions with inverse kinematics



Additional details

[Example using null space with IK]

Libraries on motion planning ([klamp't], [OMPL])

[CS 4335/5335: lectures on motion planning]

A workspace and action space should be clearly defined when performing/training on a task

Workspace: a set or region of 6D end-effector poses where the robot may perform actions. For learning, you want the most constrained workspace that still allows task completion

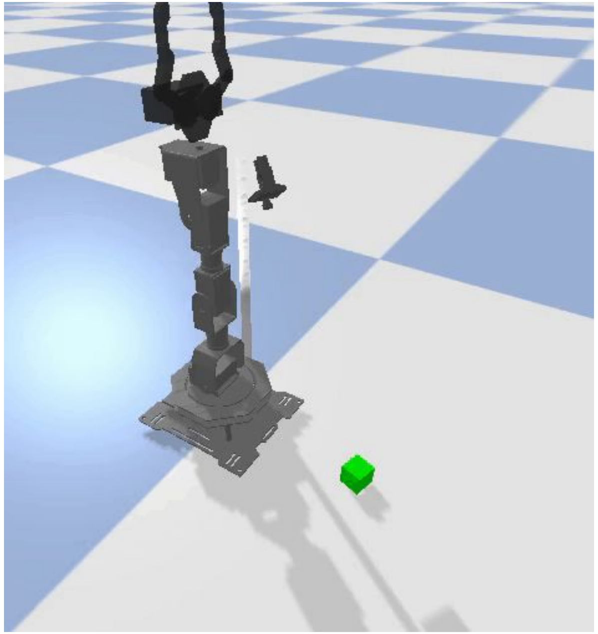
Action space: the space of possible actions that a learning agent can perform with the robot (end-effector control **and** gripper control). Choosing a good action space can greatly reduce the difficulty of learning

Example manipulation tasks

Workspace: 3D volume existing above table, extending several centimeters in the air

Action space: 6D grasp poses within workspace (no variation in gripper control)

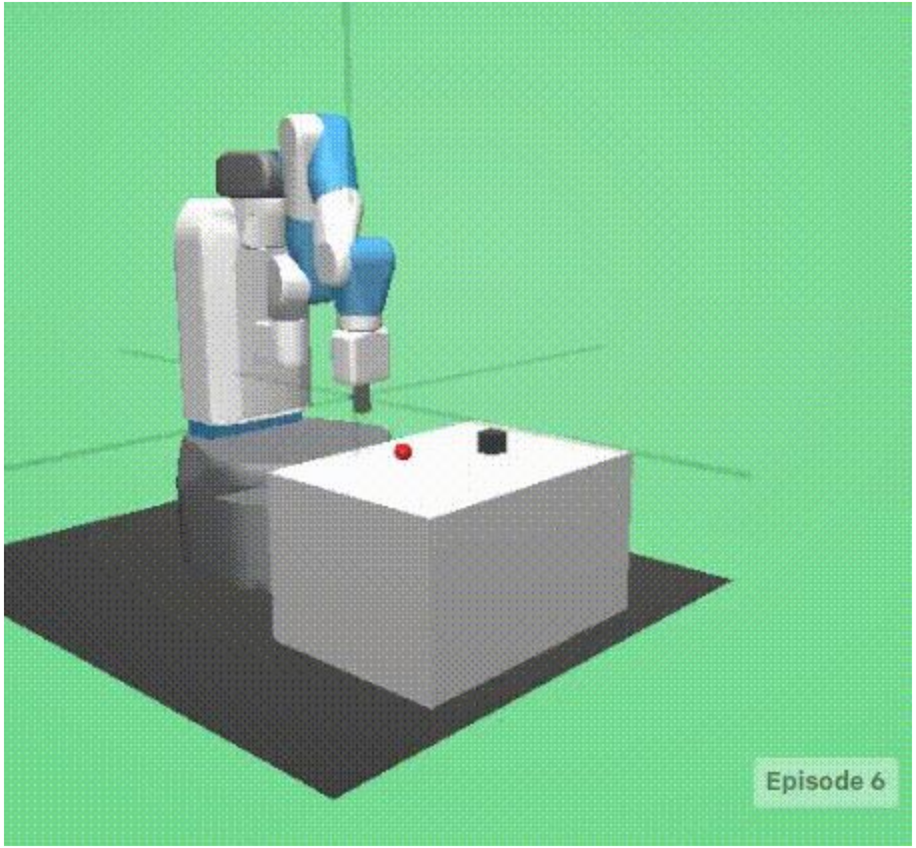
Example manipulation tasks



Workspace: 2D region in front of base of robot

Action space: x,y grasp locations (pose determined by IK, no gripper variation)

Example manipulation tasks

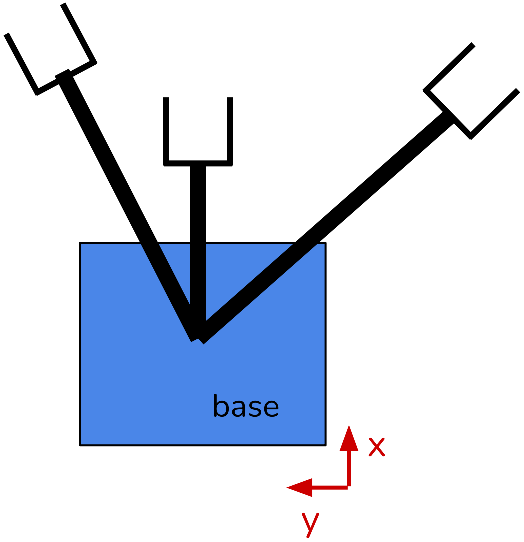


Workspace: 3D volume above table

Action space: 3D delta position control (e.g. velocity control), no control over gripper

Note about xArm

There are 5 motors in the arm, so it cannot realize all 6 dimensions of end-effector pose



We can express this restricted set of orientations using euler angles, exercise left to the reader.