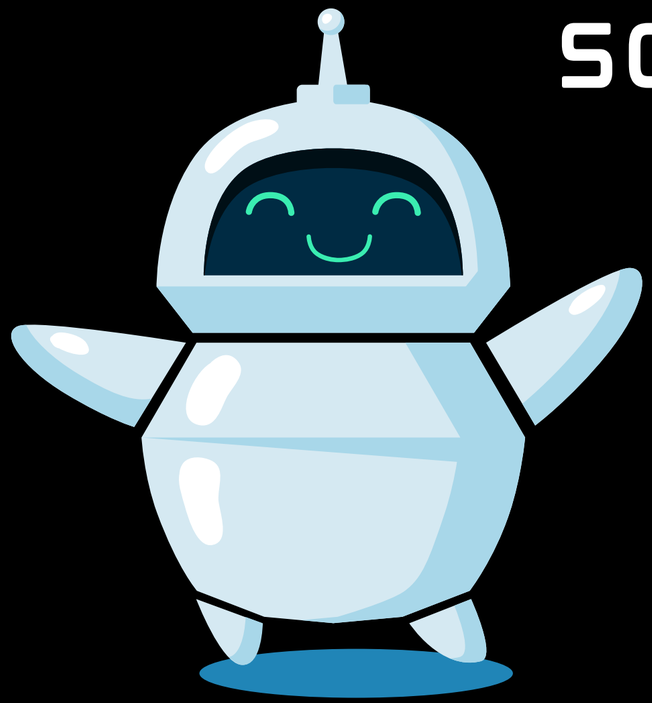
WELCOME TO
PYBULLET & CONTROL
WORKSHOP

BACICS OF TERMINAL

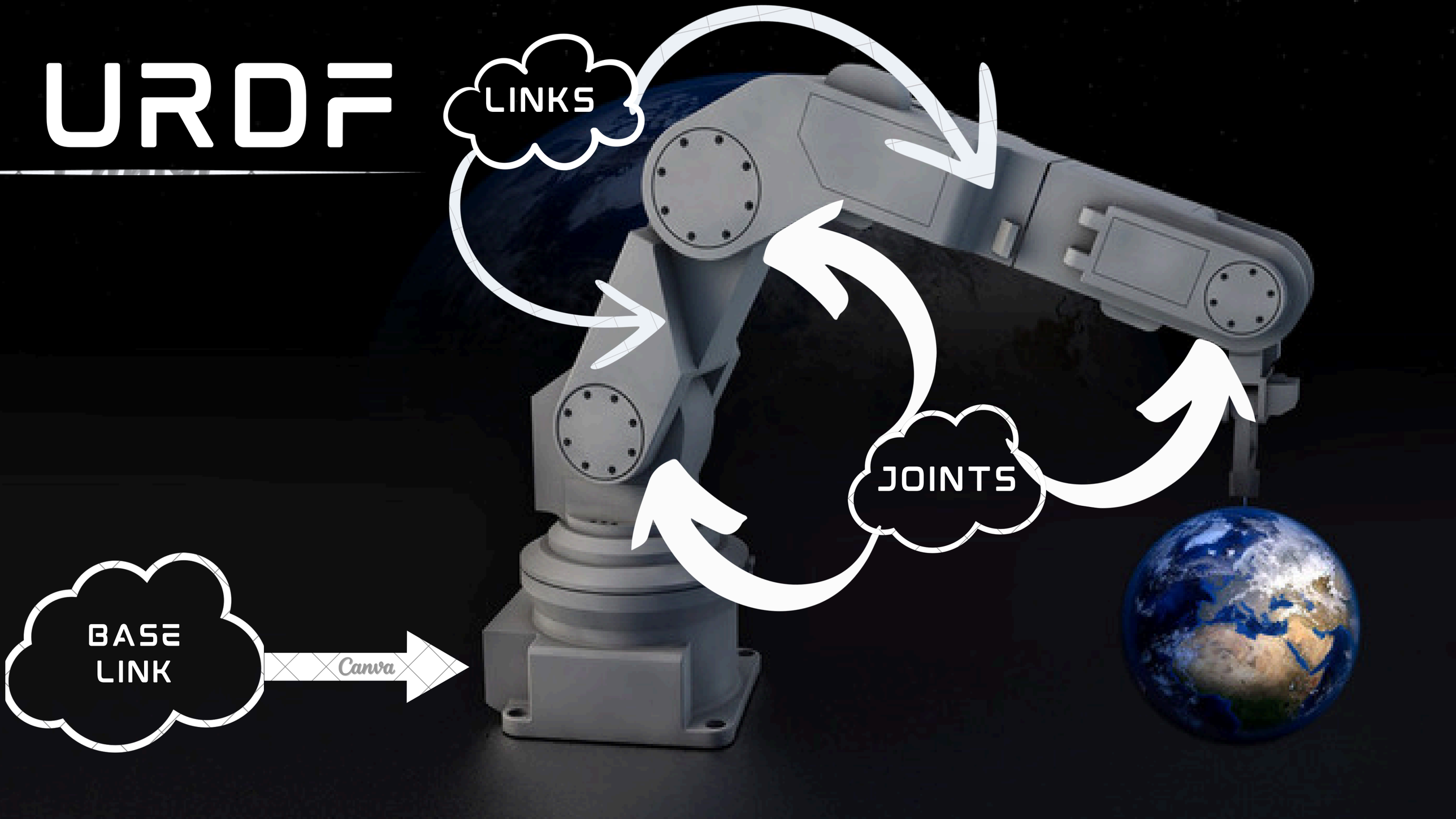
- **Up Arrow:** Will show your last command
- **Down Arrow:** Will show your next command
- **Tab:** Will auto-complete your command
- **Ctrl + L:** Will clear the screen
- **Ctrl + C:** Will cancel a command
- **Ctrl + R:** Will search for a command
- **Ctrl + D:** Will exit the terminal



SOME BASIC ANACONDA COMMANDS

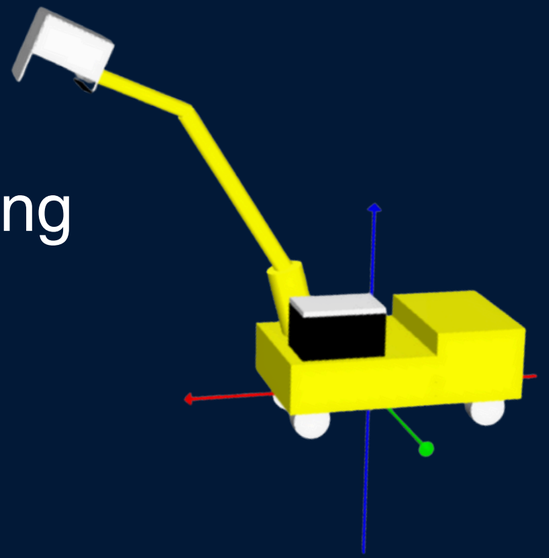
cd ~	Change to home directory
cd ..	Change to parent directory
cd [dirname]	Change directory to specific directory
ls	List directory contents
rm[filename]	Remove file

URDF



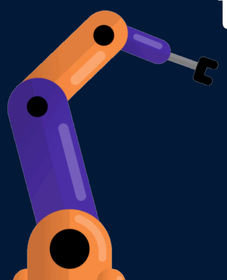
URDF

Unified Robot Description Format (URDF) is an XML file format that describes a robot's physical properties, including its joints, motors, and mass.



WHAT IT'S USED FOR

URDF is used to model multibody systems, such as robotic arms and animatronic robots. It's commonly used in the Robot Operating System (ROS) tools, such as rviz and Gazebo simulator.

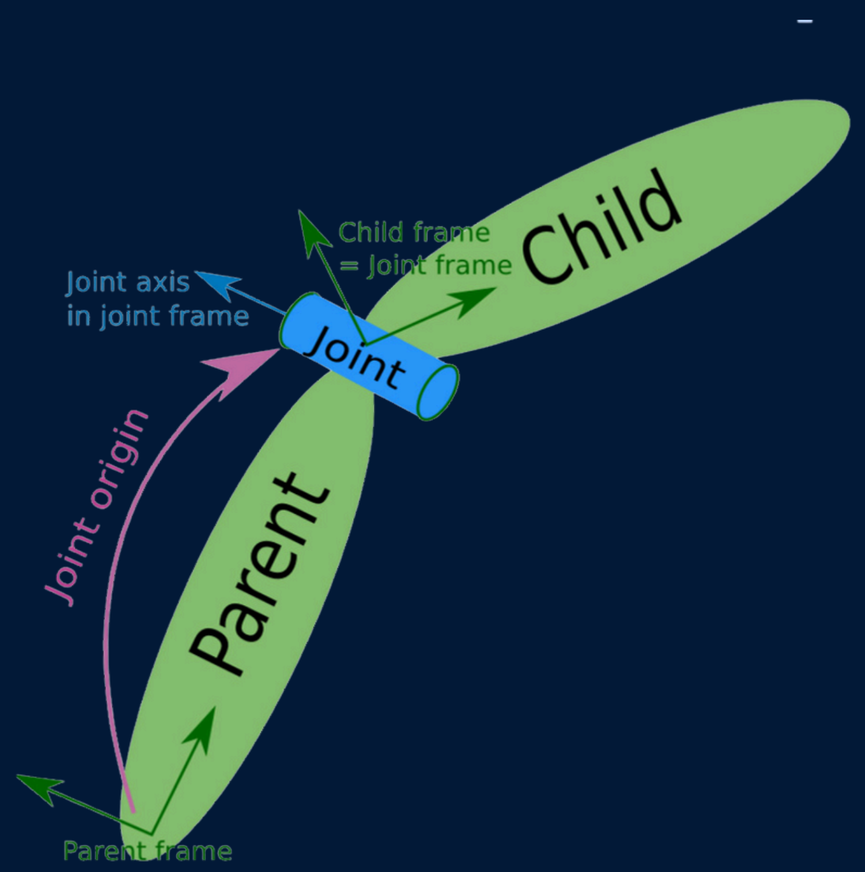


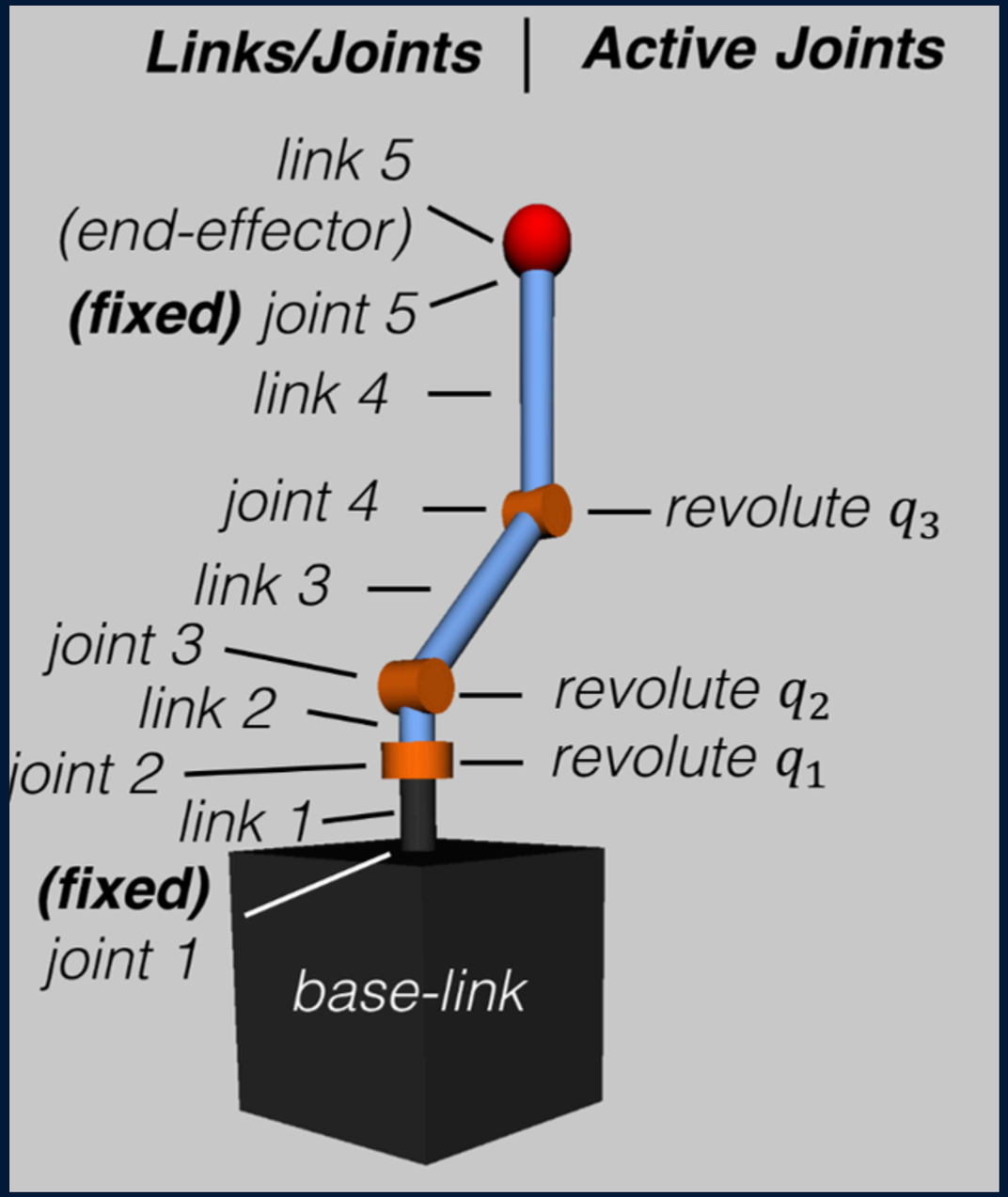
LINKS:

A link in robotics is a rigid part that connects joints in a robot, and is also known as a kinematic link or element. Links are movable and can have relative motion with other parts of the robot.

JOINTS:

Robot joints, also known as axes, are the movable connections between parts of a robot that allow it to move. They are similar to human joints and enable robots to bend, twist, and interact with their environment.

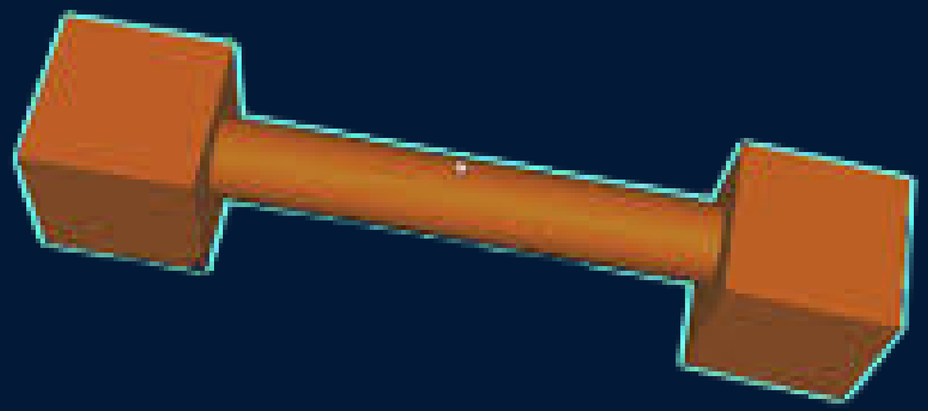



Links/Joints
Active Joints
link 5
(end-effector)
(fixed) joint 5
link 4
joint 4
revolute q_3
link 3
joint 3
revolute q_2
link 2
joint 2
revolute q_1
link 1
(fixed)
joint 1
base-link

TYPES OF JOINTS

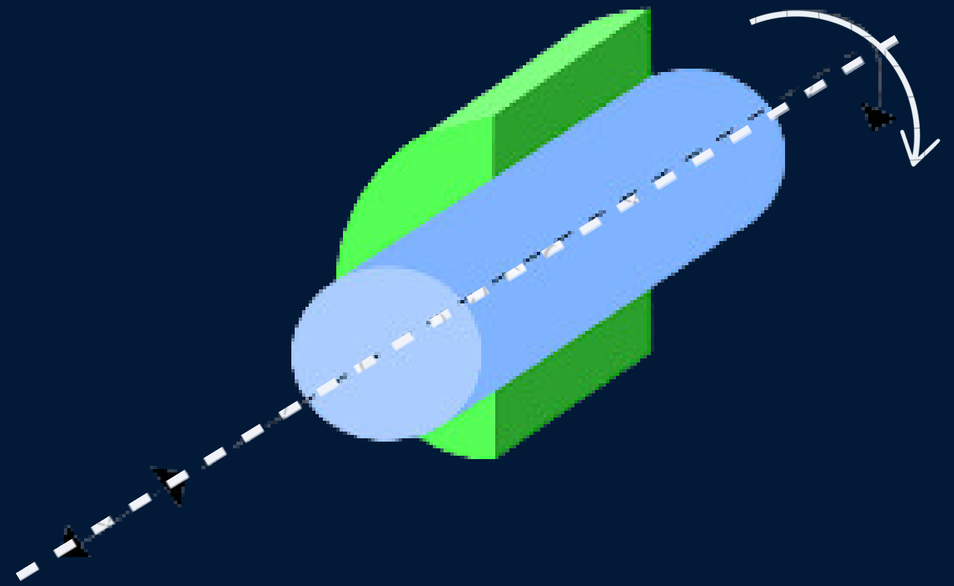
FIXED JOINT:

A"fixed joint" in robotics refers to a connection between two robot links that allows absolutely no movement between them, it is the robotic equivalent of an immovable joint in biology, like the sutures in the skull.



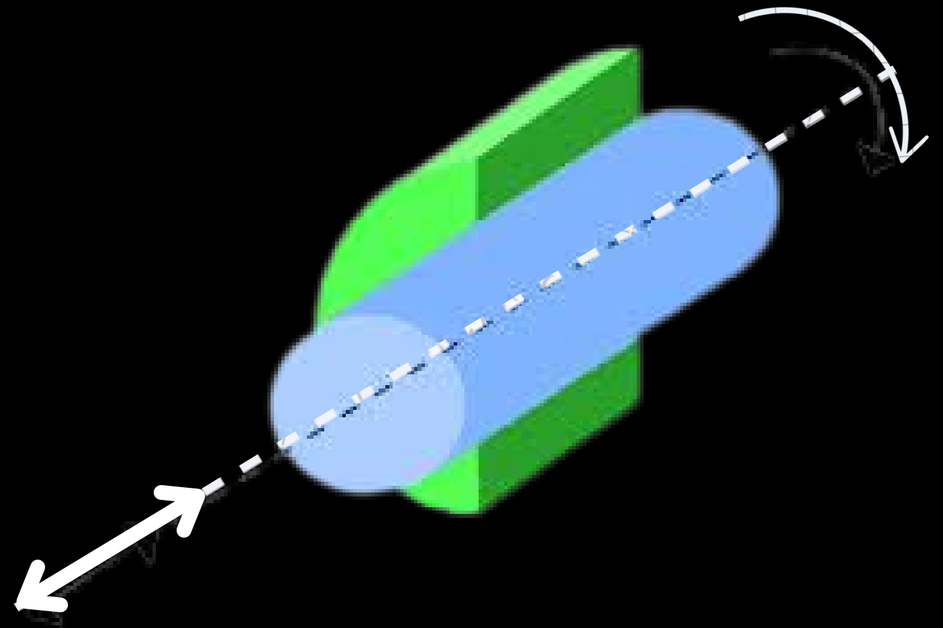
REVOLUTE JOINT:

A revolute joint, also known as a hinge joint or rotary joint, is a type of robot joint
that allows for rotation around a single axis.



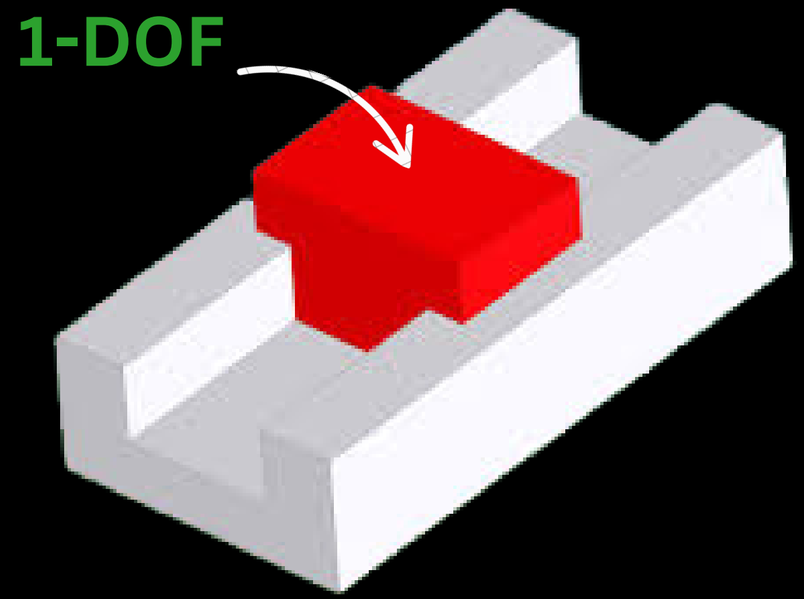
CYLINDRICAL JOINT:

A cylindrical joint in robotics is a joint that allows for both rotational and linear motion, and is often used in robots for manufacturing, packaging, and material handling.



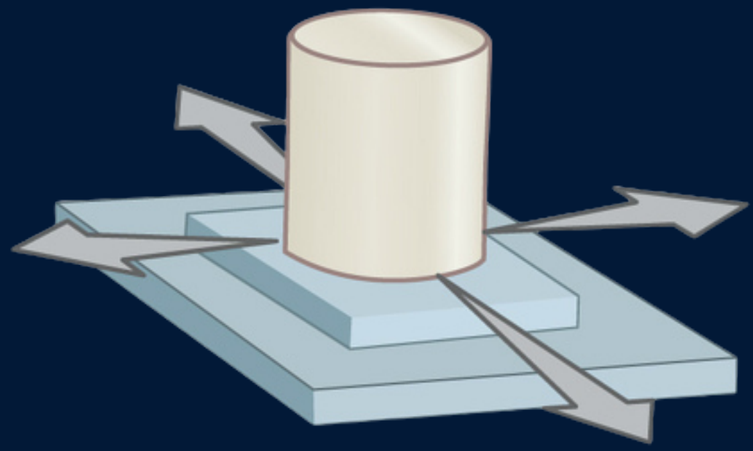
PRISMATIC JOINT:

A prismatic joint is a type of joint in robotics that allows linear motion between two links, bases, or end effectors. It's also known as a linear joint or sliding pair



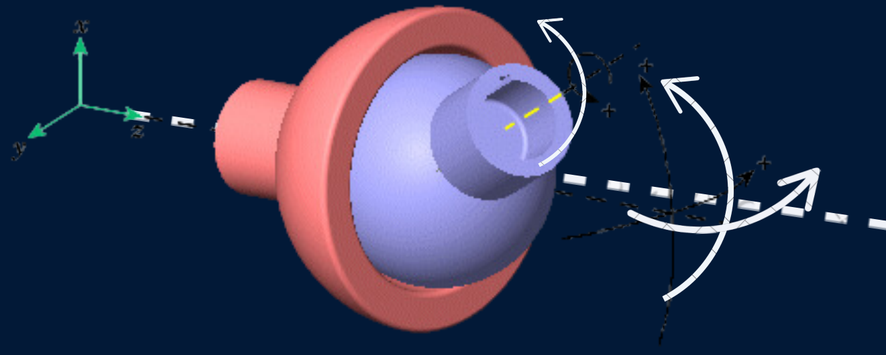
PLANAR JOINT:

A planar joint in robotics is a type of joint that allows movement within a plane, with two degrees of translational freedom and one degree of rotational freedom. Planar joints are often used in applications that require planar motion, such as painting or welding robots.



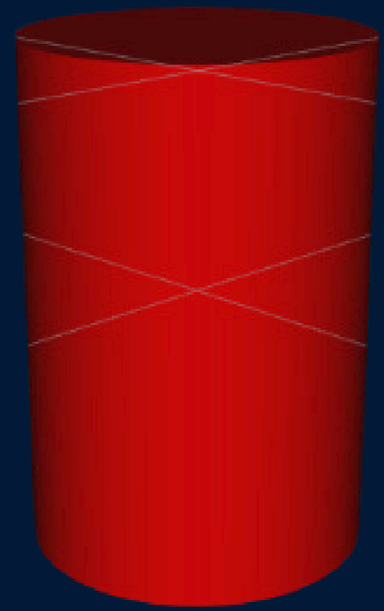
SPHERICAL JOINT:

A spherical joint allows full rotation around three axes, while a universal joint enables two-axis rotation for torque transfer, like in driveshafts. Spherical joints provide flexibility; universal joints are robust for limited motion.

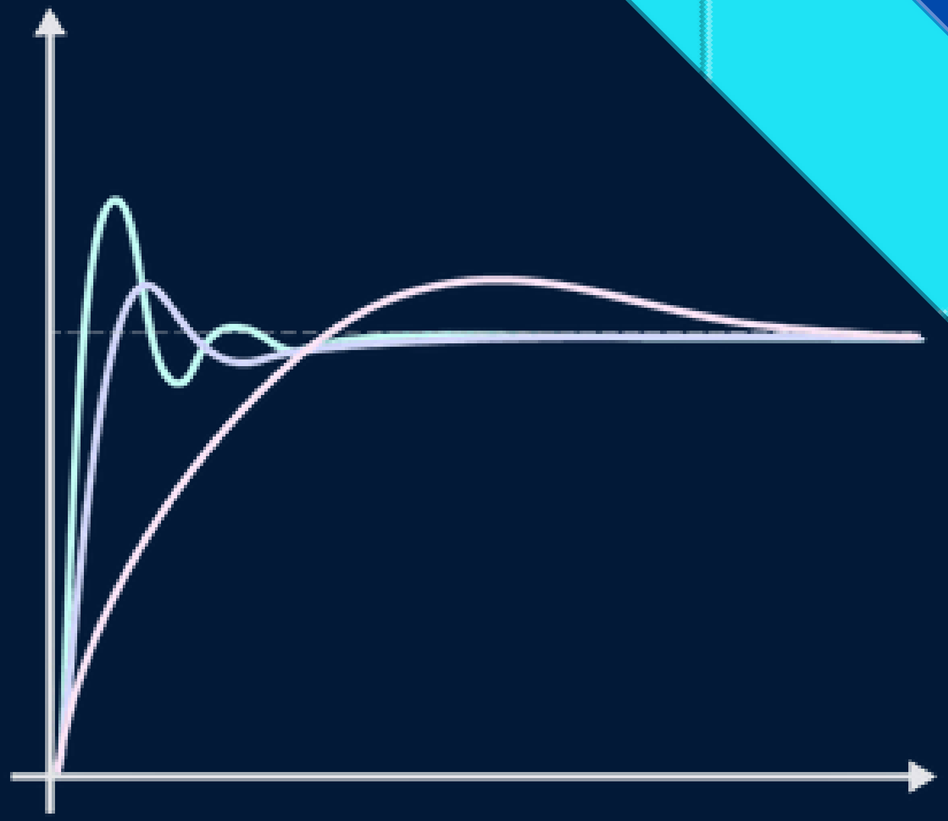


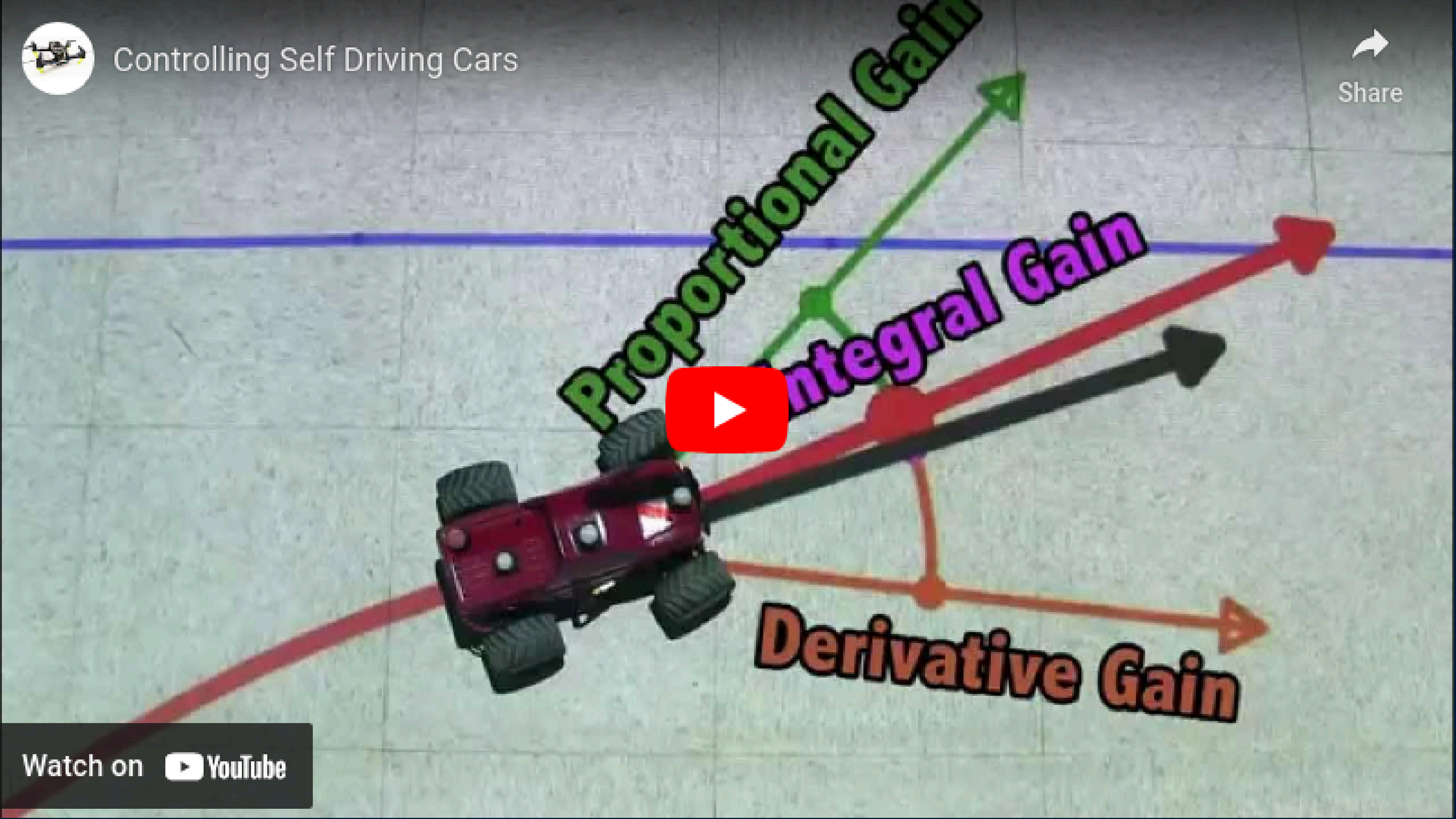
EXAMPLE

```
<?xml version="1.0"?>
<robot name="myfirst">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```



PID Controller



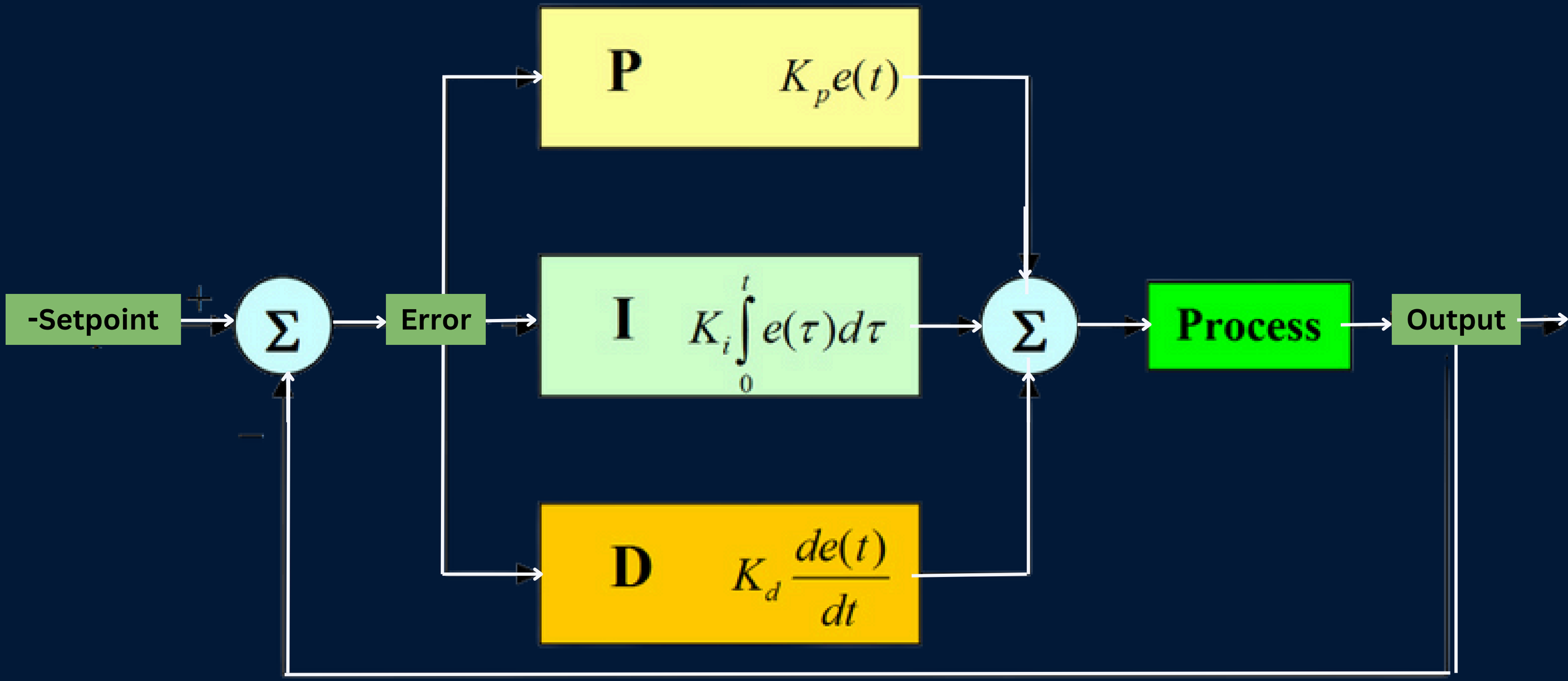

Controlling Self Driving Cars
Share
Proportional Gain
Integral Gain
Derivative Gain
Watch on YouTube

INTRODUCTION TO PID CONTROLLERS AND THEIR APPLICATIONS

A PID controller is an instrument used in industrial control applications to regulate temperature, flow, pressure, speed and other process variables. PID (proportional integral derivative) controllers use a control loop feedback mechanism to control process variables and are the most accurate and stable controller.

A proportional–integral–derivative controller (PID controller or three-term controller) is a feedback-based control loop mechanism commonly used to manage machines and processes that require continuous control and automatic adjustment.

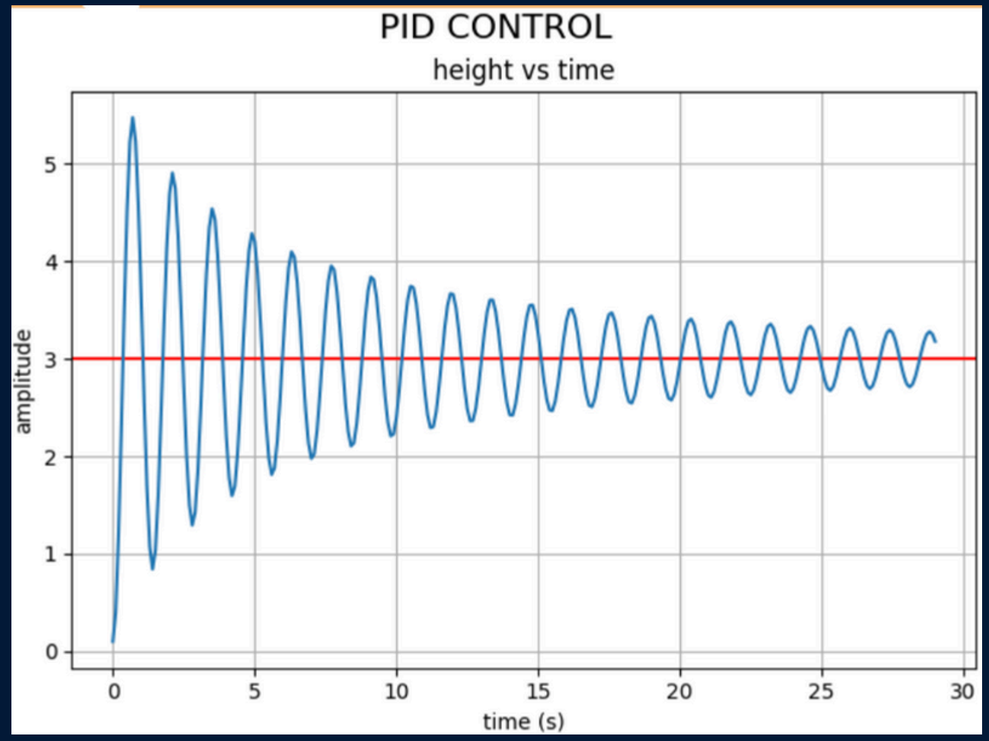
LOGIC FOR PID:



The PID controller automatically compares the desired target value (setpoint or SP) with the actual value of the system (process variable or PV). The difference between these two values is called the error value, denoted as e(t).

P(PROPORTIONAL)

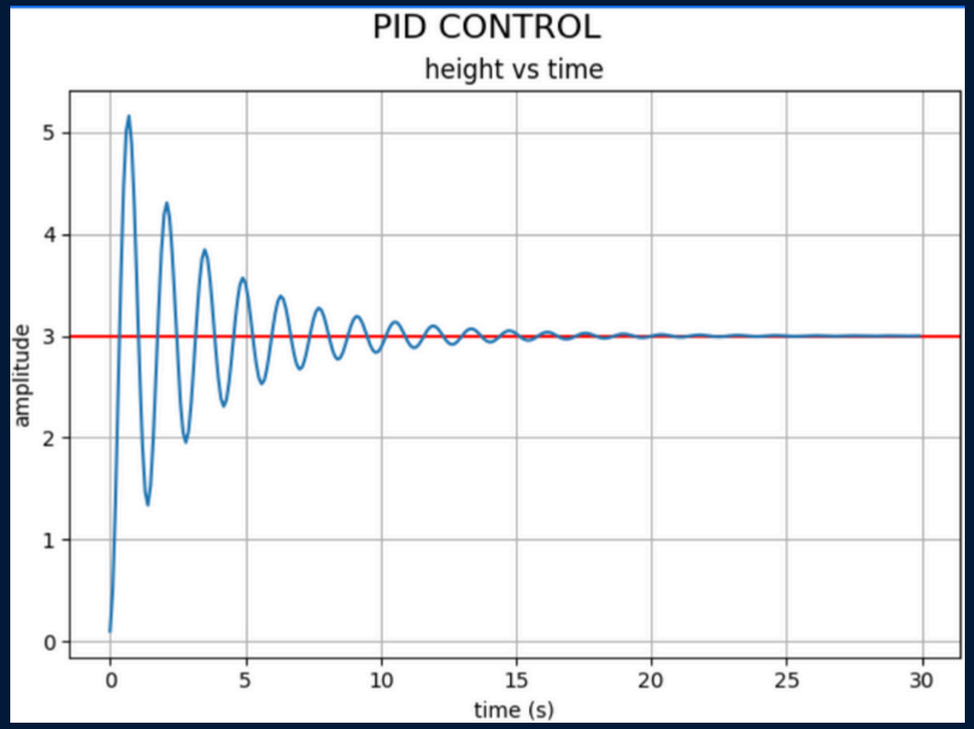
Term P is proportional to the current value of the SP – PV error e(t).
If the error is large, the control output is proportionally large, scaled by the gain factor Kp. Proportional control alone results in a steady-state offset (SP–PV) because an error is needed to produce a response, leading to equilibrium under steady conditions.



P CONTROL

D(DERIVATIVE)

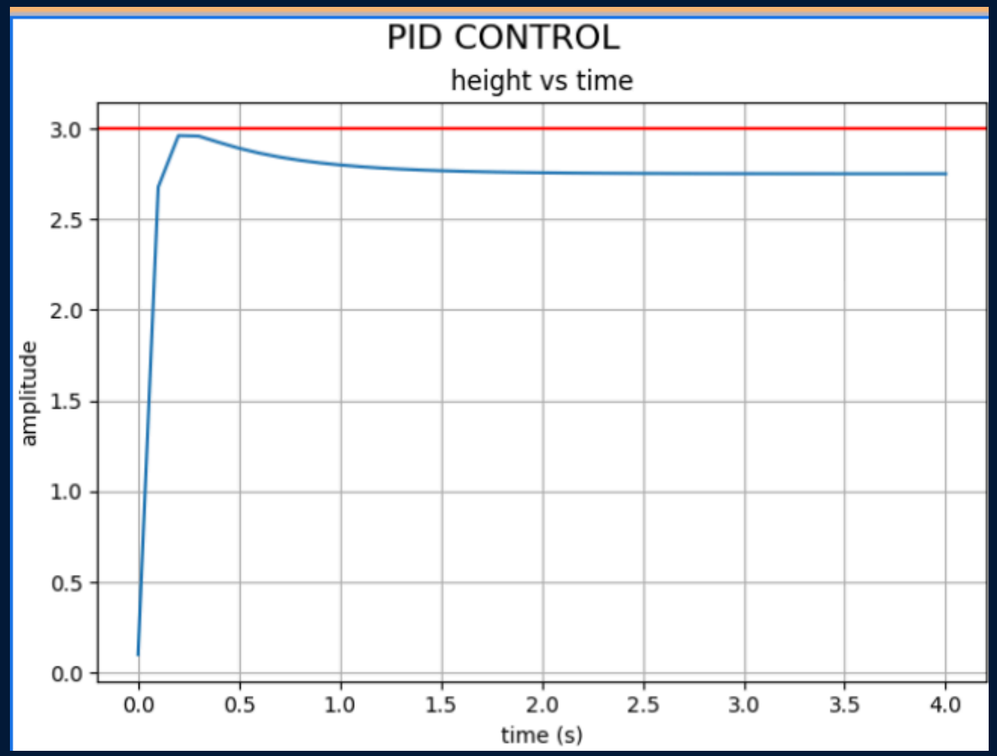
Term D is a best estimate of the future trend of the SP – PV error, based on its current rate of change. It is sometimes called "anticipatory control", as it is effectively seeking to reduce the effect of the SP – PV error by exerting a control influence generated by the rate of error change. The more rapid the change, the greater the controlling or damping effect.



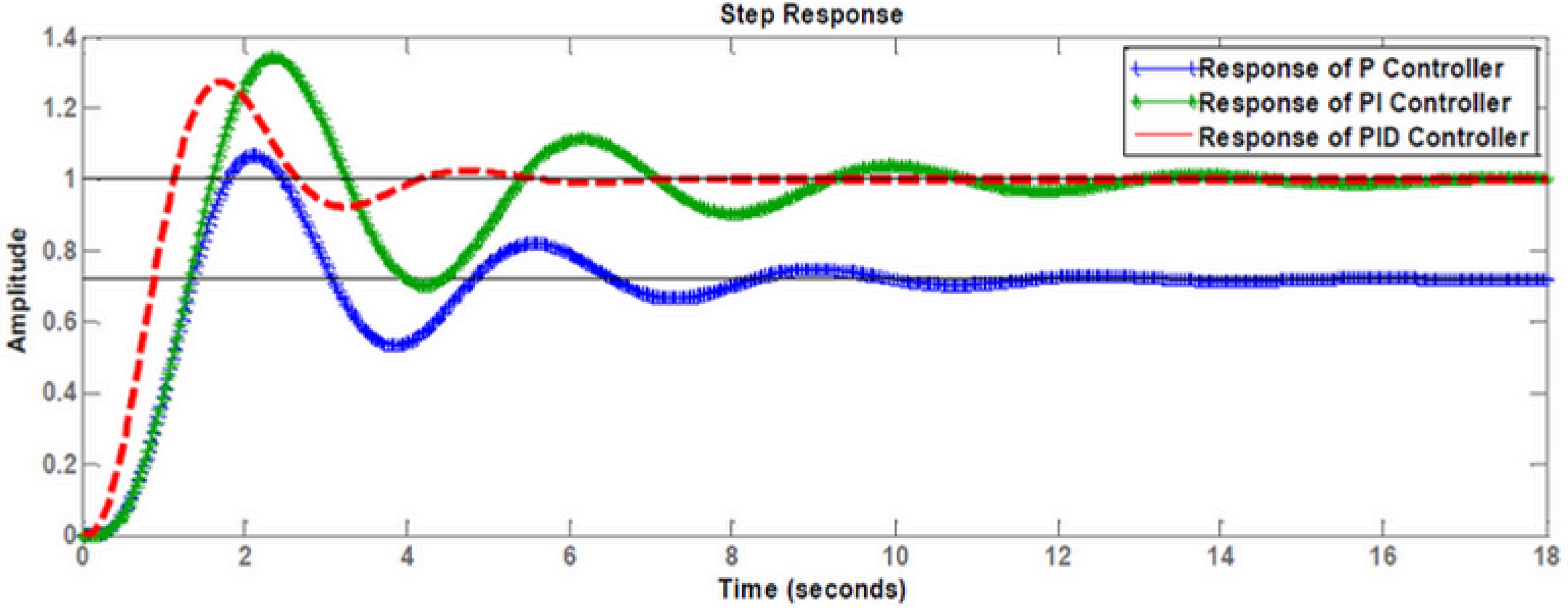
PD CONTROL

I(INTEGRAL)

Term I accounts for past values of the SP – PV error and integrates them over time to produce the I term. For example, if there is a residual SP – PV error after the application of proportional control, the integral term seeks to eliminate the residual error by adding a control effect due to the historic cumulative value of the error. When the error is eliminated, the integral term will cease to grow. This will result in the proportional effect diminishing as the error decreases, but this is compensated for by the growing integral effect.



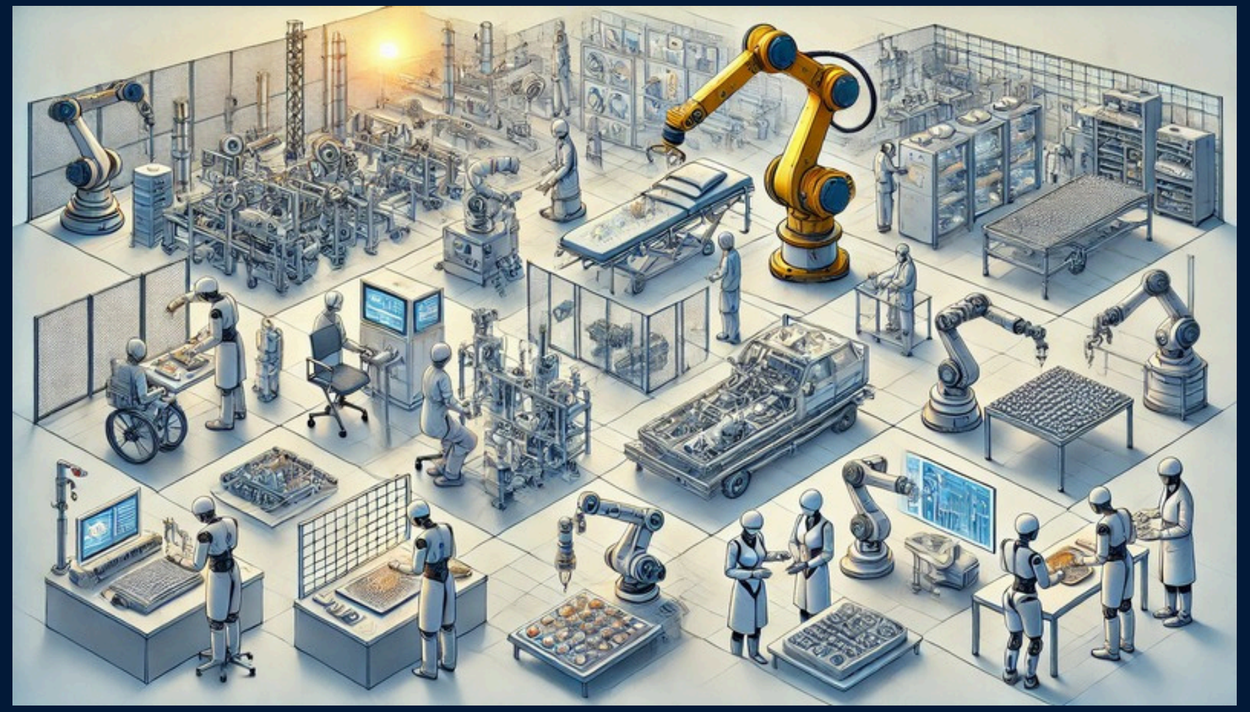
PID CONTROL


Step Response
Response of P Controller
Response of PI Controller
Response of PID Controller
Amplitude
Time (seconds)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
0
2
4
6
8
10
12
14
16
18

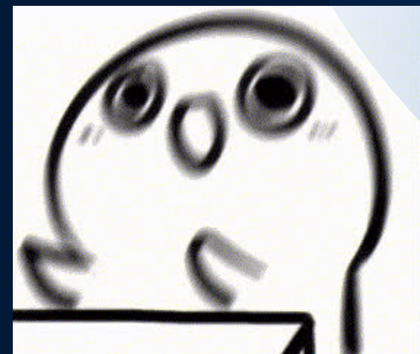
APPLICATIONS OF MOTION CONTROL

Movement control in robotics is crucial for enabling robots to perform couple of tasks with precision and accuracy, including applications like:

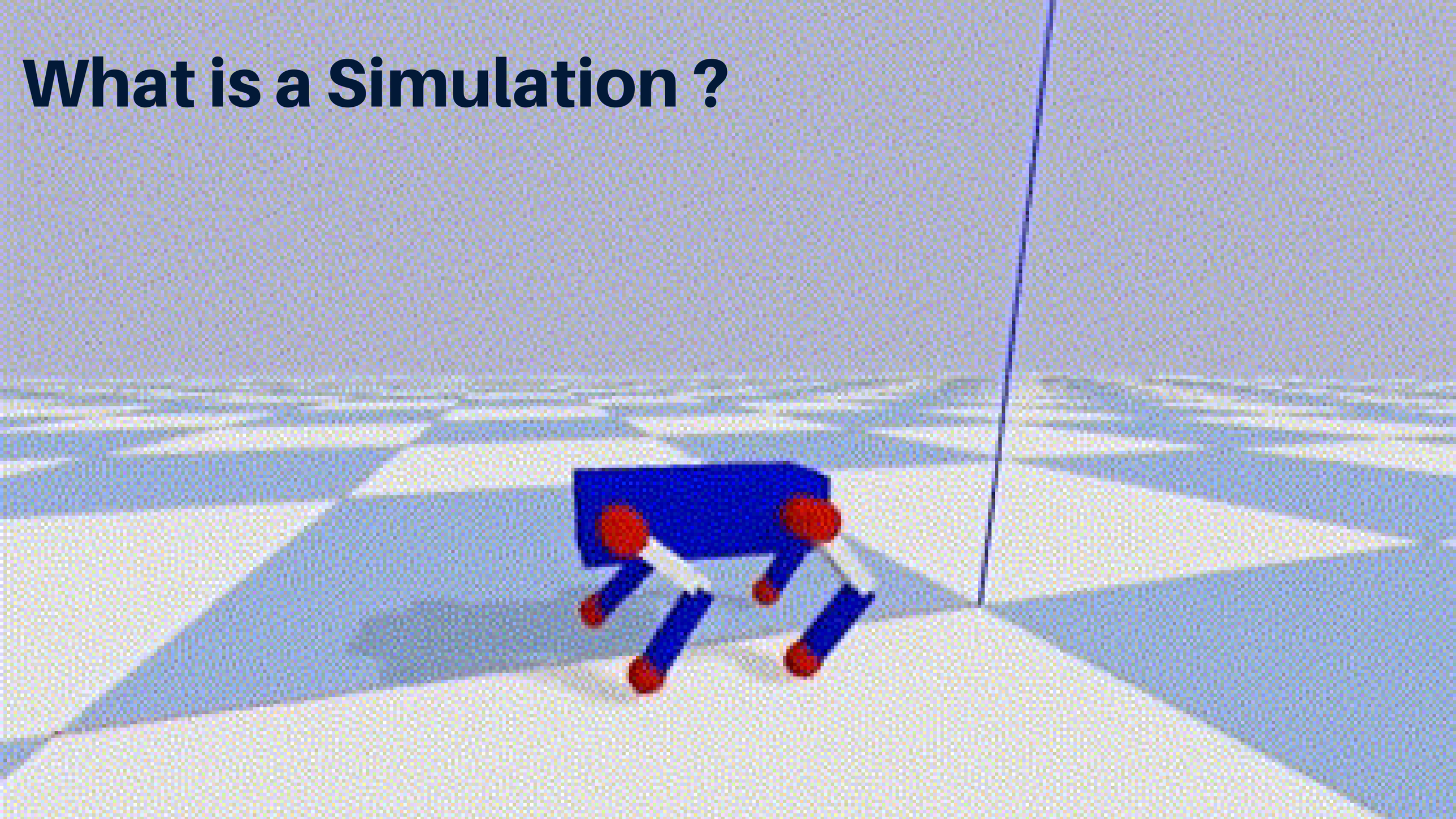
- Industrial assembly lines
- Surgical robotics
- Pick-and-place operations
- Material handling,
- 3D printing
- Automated manufacturing
- Collaborative robotics
- Food handling



INTRODUCTION TO PYBULLET



What is a Simulation ?



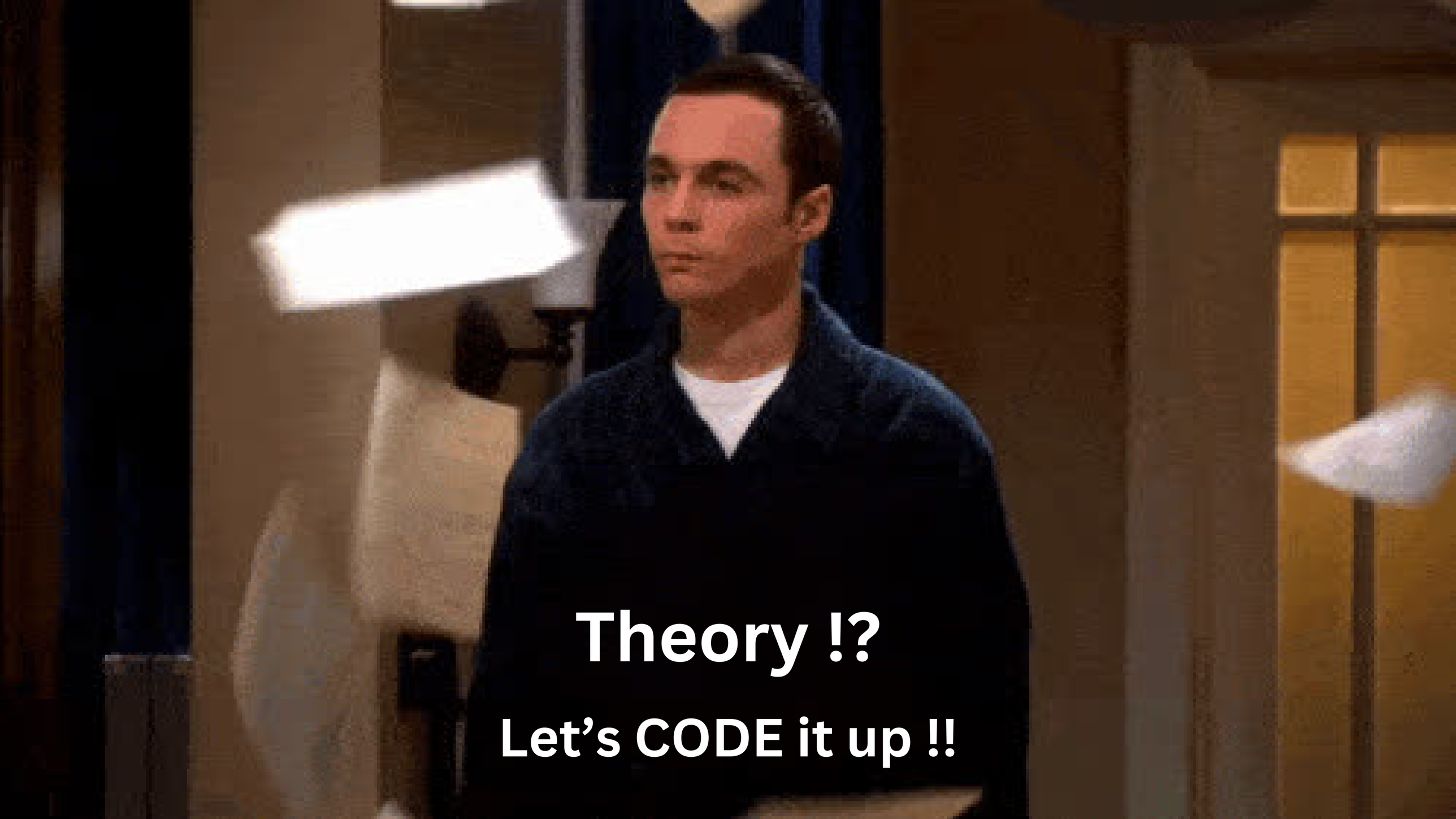
Why PyBullet ?

PyBullet is a physics engine that simulates collision detection, soft and rigid body dynamic. It is an easy to use Python module for physics simulation, robotics, etc.

• It is an open source software with an active community.

• Built for python development, hence gives more informative and clear approach for beginners.

• We don't require any external dependencies except a fully working python interpreter


Theory !?
Let’s CODE it up !!

Basic Function

1. First we import the required libraries i.e pybullet, time and pybullet_data.
2. connect

- After importing the PyBullet module, the first thing to do is 'connecting' to the physics simulation. PyBullet is designed around a client-server driven API, with a client sending commands and a physics server returning the status. PyBullet has some built-in physics servers DIRECT and GUI. Both GUI and DIRECT connections will execute the physics simulation and rendering in the same process as PyBullet
- The connect function returns a physics client id.
- The DIRECT connection sends the commands directly to the physics engine, without using any transport layer and no graphics visualization window, and directly returns the status after executing the command
- The GUI connection will create a new graphical user interface (GUI) with 3D OpenGL rendering, within the same process space as PyBullet

ox.py > ...

```python
import pybullet as p
import time
import pybullet_data
physicsClient = p.connect(p.GUI)
p.setAdditionalSearchPath
(pybullet_data.getDataPath())
p.setGravity(0,0,-10)
planeId = p.loadURDF("plane.urdf")
cubeStartPos = [0,0,1]
cubeStartOrientation = p.getQuaternionFromEuler([0,0,0])
boxId = p.loadURDF("cube.urdf"
                   ,cubeStartPos,
                     cubeStartOrientation)
for i in range (10000):
    p.stepSimulation()
    time.sleep(1./240.)
cubePos, cubeOrn = p.getBasePositionAndOrientation(boxId)
print(cubePos,cubeOrn)
p.disconnect()
```

3.setAdditionalSearchPath is used to add pybullet data to the path which contains many examples, urdf files, etc

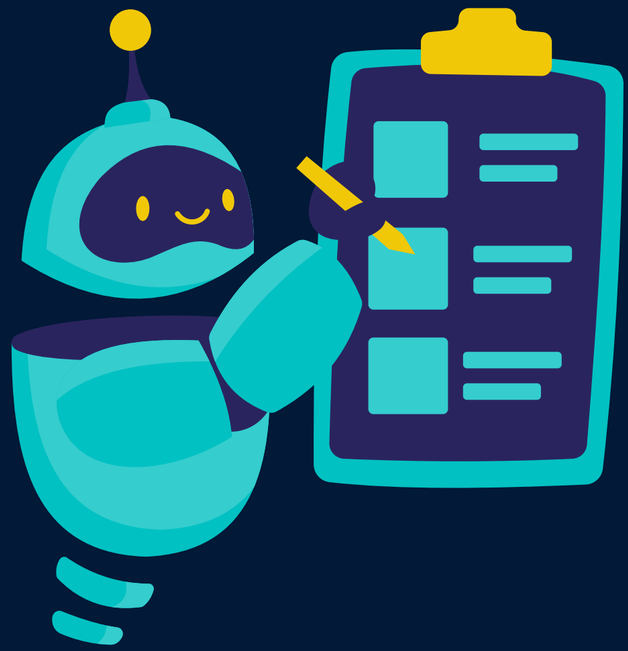
4. setGravity By default, there is no gravitational force enabled setGravity lets you set the default gravity force for all objects. The setGravity input parameters are (no return value)

5. LoadURDF The loadURDF will send a command to the physics server to load a physics model from a Universal Robot Description File (URDF)

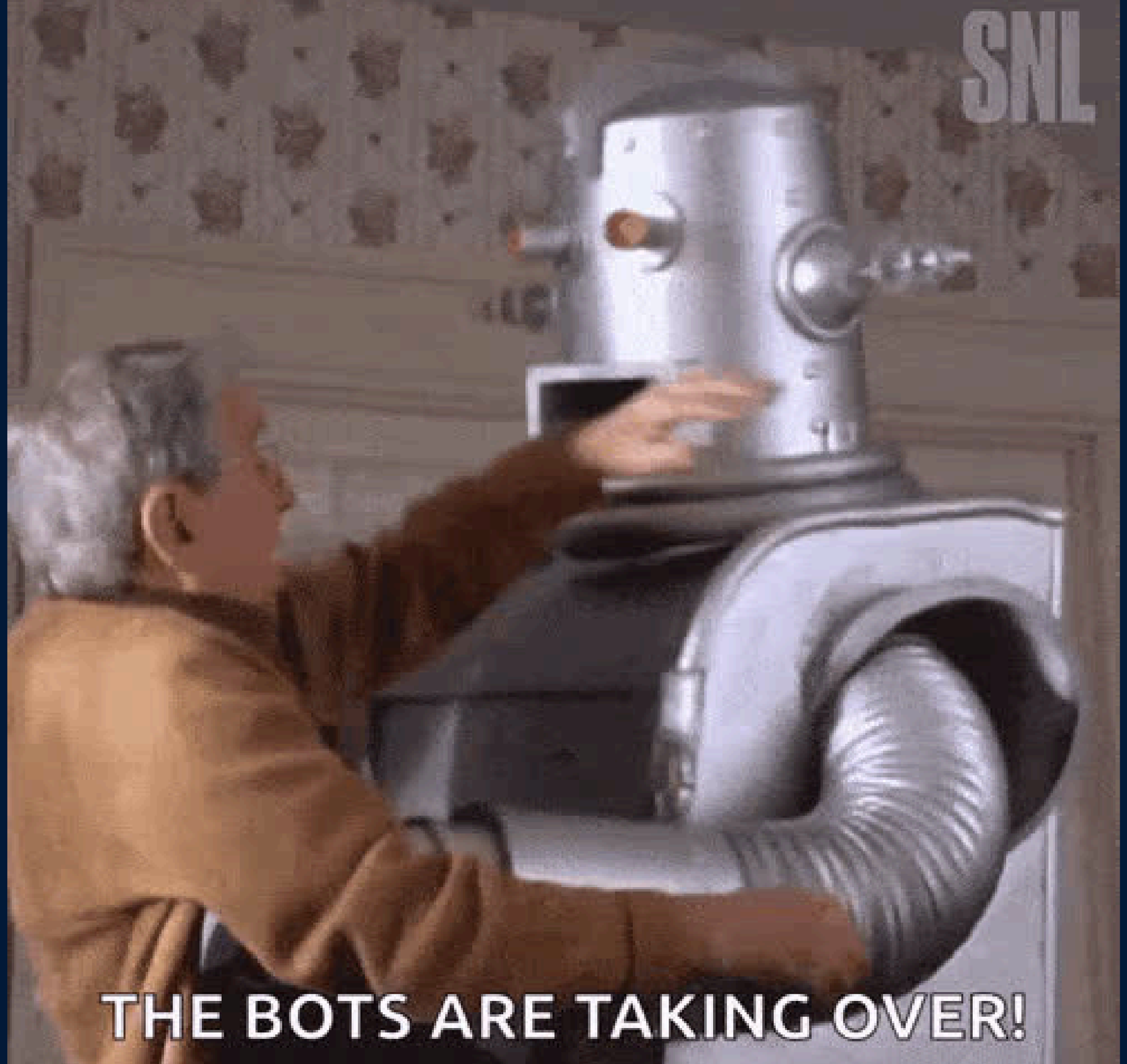
6. We store the initial position and Orientation of our urdf file in the variable cubsStartPos and cubeStartOrientation.

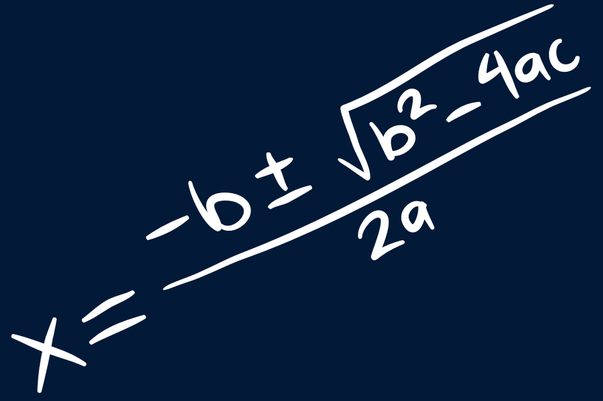
7. We import our cube urdf file in the desired position and orientation.

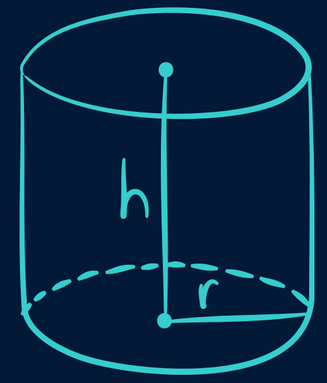
8. stepSimulation:stepSimulation will perform all the actions in a single forward dynamics simulation step. The default timestep is 1/240 second, it can be changed using the setTimeStep or setPhysics EngineParameter API.



- More examples of URDF

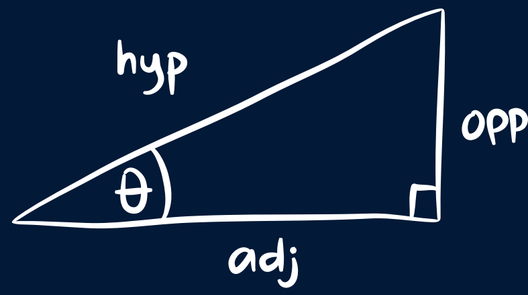

SNL
THE BOTS ARE TAKING OVER!





$V = \pi r^2 h$

Controlling a Robot



$\sin(\theta) = \frac{opp}{hyp}$

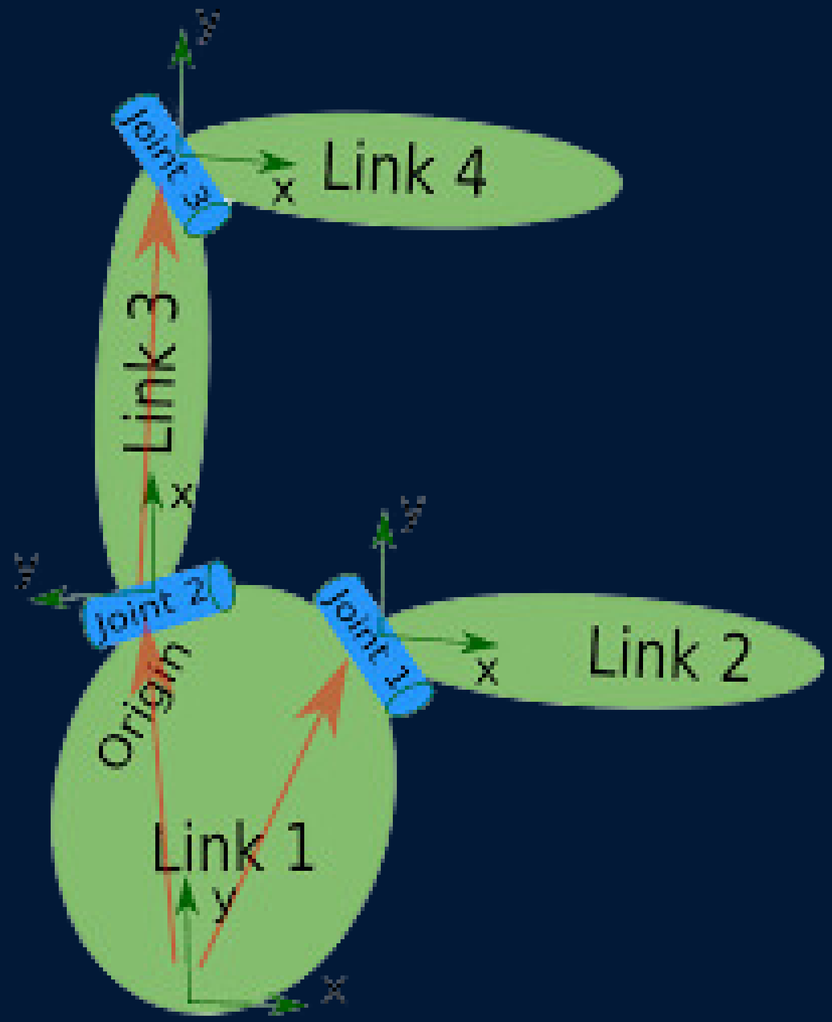
- Controlling a robot involves controlling the **JOINTS** and **LINKS** of the robot in accordance to the task it has to perform.

- Like the bones in a human arm, movable segments in a robot are linked up with robotic joints. Each joint of a robot be it revolute joint or prismatic joint is motorized.

- In simulation, it involves providing input commands or programming **control algorithms** to guide the robot's behavior while observing how it responds in a simulated world.

- In PyBullet, it can be achieved by using various prebuilt commands like **getNumJoints, getJointInfo, setJointMotorControl2/Array, getJointState(s), resetJointState** etc.

- NOTE - you can find more functions HERE



- **getJointInfo()**

We use getJointInfo() to figure out what joint IDs correspond to the joints we want to control.

input parameters:

bodyUniqueId	int	the body unique id, as returned by loadURDF etc.
jointIndex	int	an index in the range [0 .. getNumJoints(bodyUniqueId))

- **getNumJoints()**

We use getNumJoint() to find the number of joints in the robot.

input parameters:

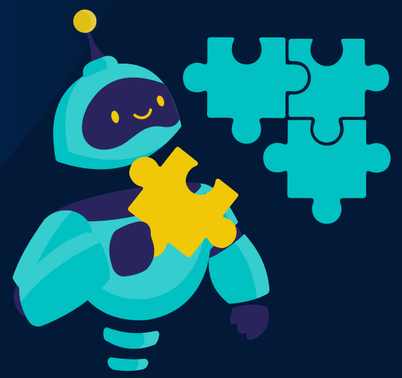
bodyUniqueId	int	the body unique id, as returned by loadURDF etc.

• setJointMotorControl()

setJointMotorControl2() allows us to change the velocity, position, or apply a torque to a joint. This is the main method used to control robots. It takes both a robot ID and a joint ID.

Input arguments:

bodyUniqueId	Int	body unique id as returned from loadURDF etc.
jointIndex	Int	link index in range [0..getNumJoints(bodyUniqueId) (note that link index == joint index)
controlMode	Int	POSITION_CONTROL (which is in fact CONTROL_MODE_POSITION_VELOCITY_PD), VELOCITY_CONTROL, TORQUE_CONTROL and PD_CONTROL.
targetPosition	float	in POSITION_CONTROL the targetValue is target position of the joint
targetVelocity	float	in VELOCITY_CONTROL and POSITION_CONTROL the targetVelocity is the desired velocity of the joint, see implementation note below.
force	float	in POSITION_CONTROL and VELOCITY_CONTROL this is the maximum motor force used to reach the target value. In TORQUE_CONTROL this is the force/torque to be applied each simulation step.



Modes of Control

Position Control Mode

p.POSITION_CONTROL

Velocity Control Mode

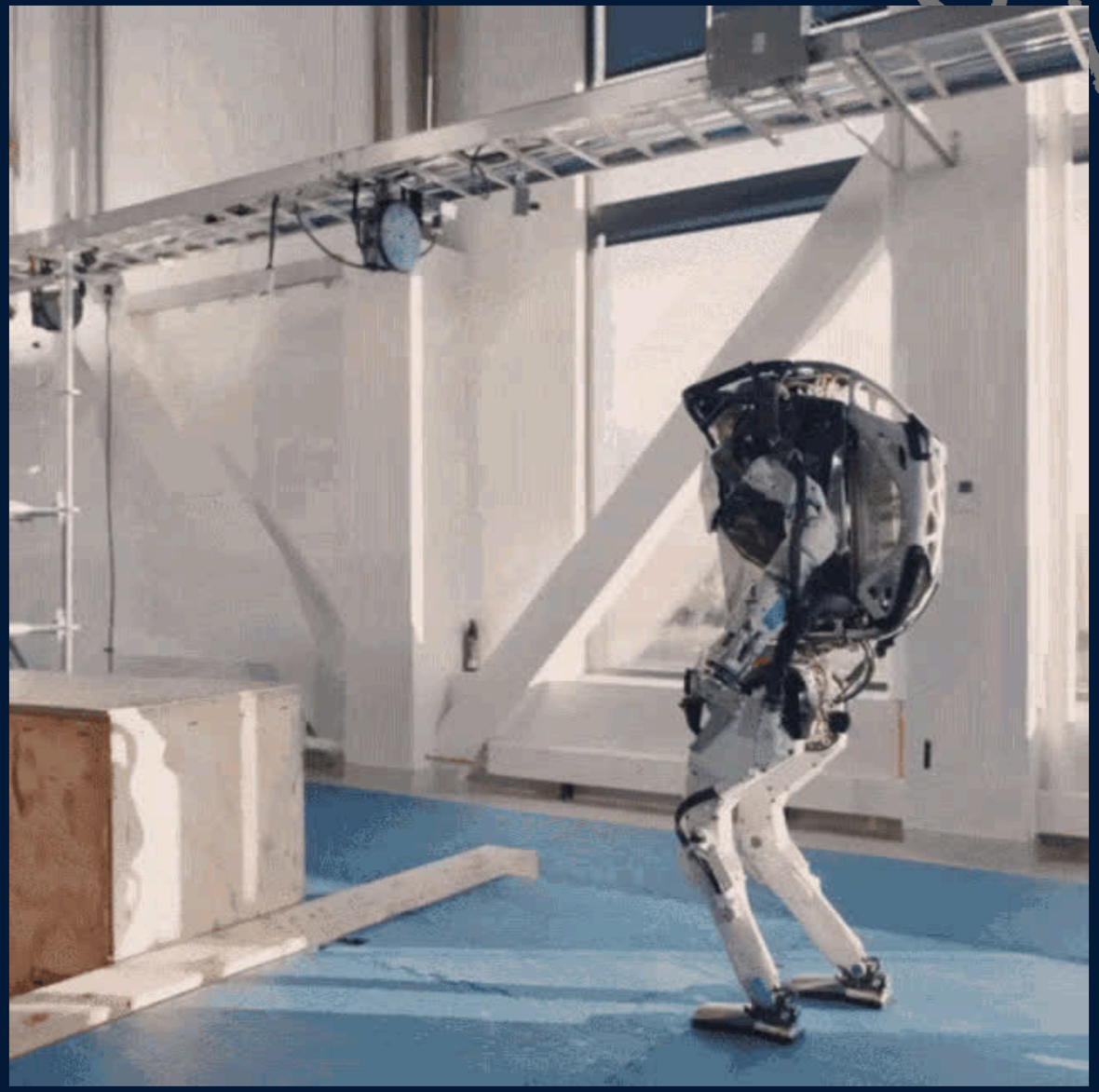
p.VELOCITY_CONTROL

Torque Control Mode

p.TORQUE_CONTROL


Happy controlling

Dynamics



• getDynamicsInfo()

You can get information about the mass, center of mass, friction and other properties of the base and links.

Input Parameters:

bodyUniqueId	int	Object unique id, as returned by loadURDF etc
linkIndex	int	Link (joint) index or -1 for the base

Example:

```
p.connect(p.GUI)
p.setAdditionalSearchPath(pybullet_data.getDataPath())

planeId=p.loadURDF("plane.urdf", useFixedBase=True)
sphereId=p.loadURDF("sphere2.urdf", basePosition=[0, 0, 5])

get_Info = p.getDynamicsInfo( bodyId, -1)
print( get_Info )

p.setGravity(0, 0, -9.8)

while(1):
  p.stepSimulation()
  time.sleep(1./240.)
```

• changeDynamics()

You can change the properties such as mass, friction and restitution coefficients using changeDynamics.

Input Parameters:

bodyUniqueId	int	Object unique id,as returned by loadURDF etc
linkIndex	int	Link (joint) index or -1 for the base
mass	double	Mass in kg
lateral_friction	double	Friction coefficient
restitution	double	Coefficient of restitution
Rolling friction	double	Rolling friction coefficient orthogonal to contact normal
Spinning friction	double	Spinning friction coefficient around contact normal
Contact damping	double	Damping of contact constraints

Example:

```
planeId=p.loadURDF("plane.urdf", useFixedBase=True)
sphereId=p.loadURDF("sphere_small.urdf", basePosition=[0, 0, 5])

p.changeDynamics(sphereId,
                 -1,
                 lateralFriction=0,
                 restitution=1,
                 linearDamping=0)

p.changeDynamics(planeId,
                 -1,
                 lateralFriction=0,
                 restitution=1,
                 linearDamping=0)
p.setGravity(0, 0, -9.8)

while(1):
  p.stepSimulation()
  time.sleep(1./240.)
```


OMG ITS SO
DYNAMIC

All Good Things must Come to an End!