NURO Arm

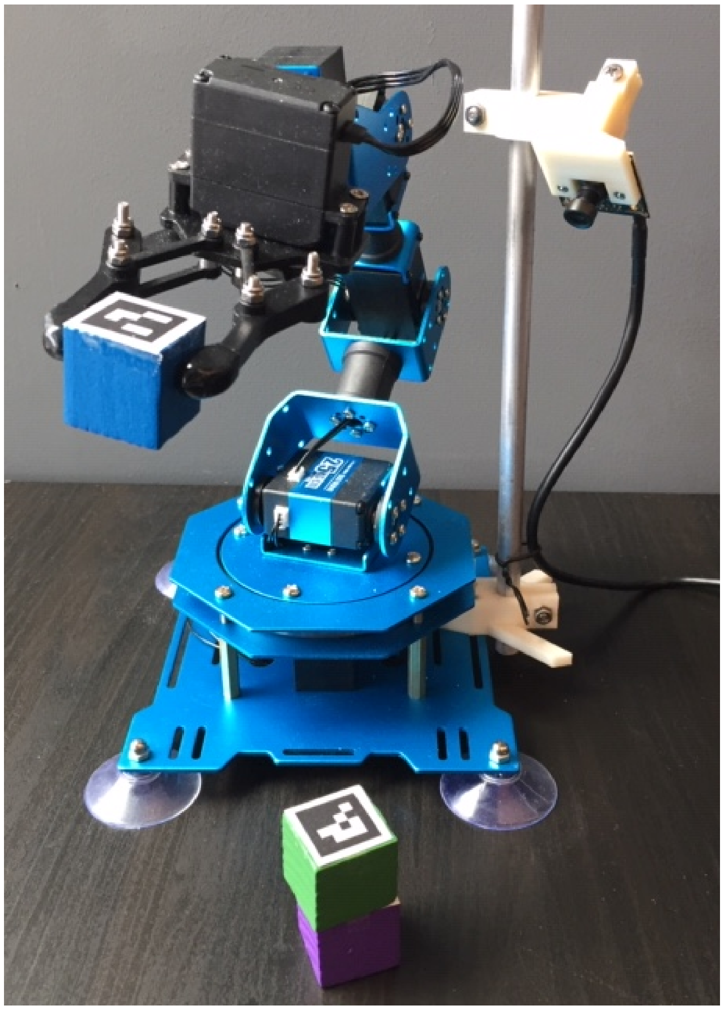
Release 0.0.1

David Klee

Jul 12, 2022

CONTENTS

nuro_arm is a great introduction to programming robots for high school and early-college students. The API provides intuitive, high-level commands to control a low-cost robotic arm. For those unable to purchase the robot, we offer a simulator that works on any operating system. We believe *nuro_arm* is a valuable educational tool for hands-on learning of concepts ranging from writing for-loops to designing reinforcement learning algorithms.



CHAPTER

ONE

GETTING STARTED

1.1 Installing Software

1.1.1 Mac

Install the repository using pip.

```
pip install nuro-arm
```

1.1.2 Windows

We recommend installing pybullet with conda first, then install the repository with pip.

```
conda install -c conda-forge pybullet

pip install nuro-arm
```

Now, you have to download the HID library (needed for communicating with robot over USB):

1. Download zip file
2. In File Explorer, navigate to *[downloads directory]\hidapi-win\x64*
3. Copy the files: "hidapi.dll", "hidapi.lib", "hidapi.pdb"
4. Paste them in *C:\Users\[username]\Anaconda3\envs\[your-env-name]*

1.1.3 Linux

First, install the repository using pip.

```
pip install nuro-arm
```

Then, install the following libraries which are needed to communicate with the robot over the USB connection.

```
sudo apt-get install libhidapi-hidraw0 libhidapi-libusb0
```

If you experience problems connecting to the robot, you may need to run the following command (usually everytime you restart the machine).

```
sudo service fwupd stop
```

1.2 Parts List

The full robot kit costs around $250. The robot and camera can be ordered easily on Amazon. Currently, the camera is mounted to the robot base using custom 3D printed parts (STL files). We are working on sourcing a camera mount that does not require 3D printing.

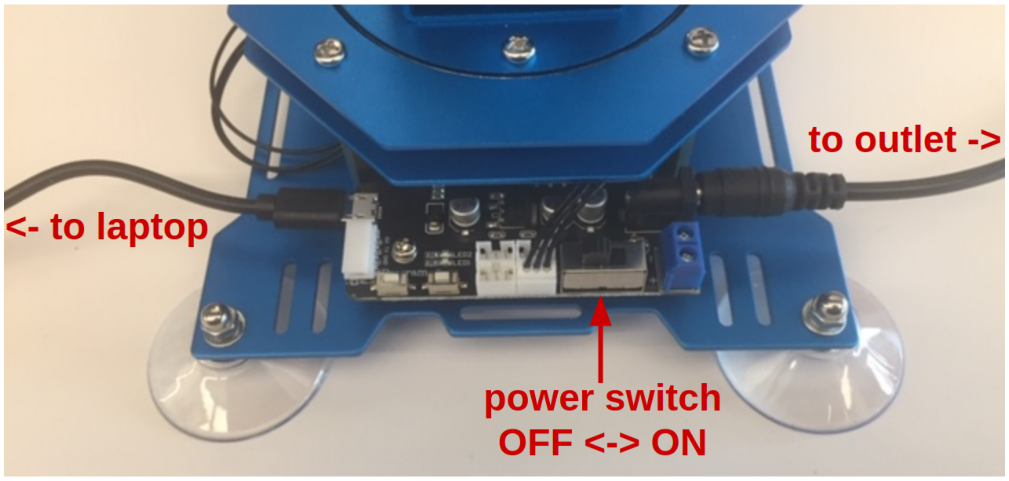
Description	Price
Hiwonder xArm	$200
USB Camera	$32
Toy Cubes	$20
Camera Stand	$10

1.3 Assembly Instructions

The assembly process takes about 2.5 hours and does not require any additional tools. The parts will all be located in the Hiwonder box. To assemble the robot, watch the following instructional videos provided by Hiwonder.

1. Assembly 01
2. Assembly 02: self-tapping means the screws will have a pointy end.
3. Assembly 03: it may take some force to fit the blue parts over the servo horns; the portion after 3:25 is about cable management and is optional. If any wires are sticking out too much, you might want to fasten them down to avoid them getting caught during motion.

The final step is to plug in the robot. We must plug two things into the controller board on the robot: the power supply and the usb cable. See the pictures below to understand how to plug things in. If everything is correct, when you flip the power switch to ON, lights on the motors will turn on. If you hear a beeping noise, this means the power supply is not plugged in. Do not leave the robot on for extended periods of time, so make sure to flip the power switch when done.



1.4 Calibration

The robot needs to be calibrated after assembly. Initiate the calibration process with the following command, and you will be guided through the process with several popup windows.

```
calibrate_xarm
```

CHAPTER

TWO

USING ROBOT ARM

2.1 Connecting to Robot

Before connecting to the robot, turn it ON using the switch on the control board. If it makes a beeping noise, then you need to plug in the power cable. Do **not** run the commands on this page unless you have already *calibrated the robot*.

To interface with the robot, use the `RobotArm` class. You can choose between using the real xArm robot (`controller_type='real'`) or a simulator version of the robot (`controller_type='sim'`) if you do not have the robot. This is shown in the following Python code snippet. Notice the last two commands, which show how to change from **active mode** to **passive mode**. The robot defaults to active mode (all motors are on and will resist movement). If you plan to move the robot by hand (for instance to demonstrate movements), then you must enter passive mode. The robot will automatically disconnect when the script ends.

```python
from nuro_arm import RobotArm

# connect to robot
controller_type = 'real' # or 'sim'
robot = RobotArm(controller_type)

# turn off motors, useful for guiding robot by hand
robot.passive_mode()

# turn on motors
robot.active_mode()
```

2.2 Joint Angle Control

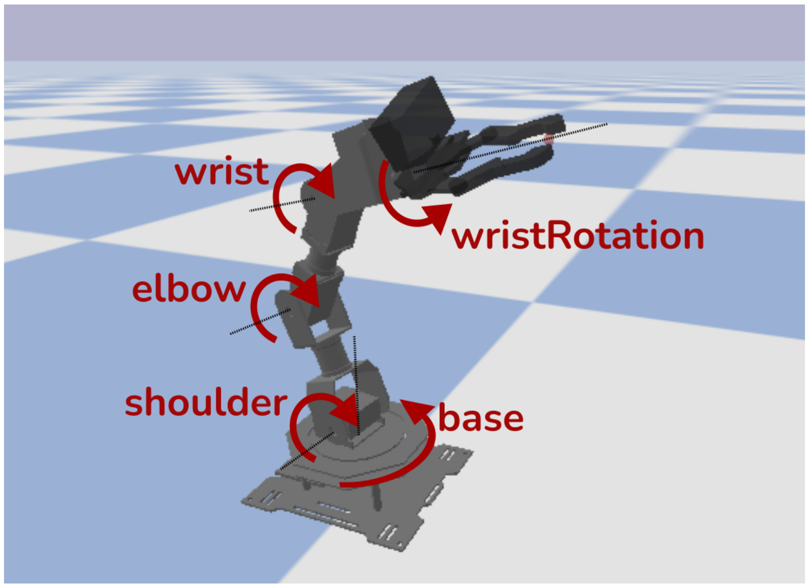
The simplest way to control the robot is to set the angle for each of the five joints in the arm. The names and directions of each joint are shown in the image below. For instance, by increasing the angle of the elbow joint, then the robot's gripper is moved closer to the ground. If all joint angles are set to 0, then the robot is in the HOME position (this is what you set during calibration). The robot expects joint angles in radians.

If you want to gain more intuition about the joints of the robot, try running the script: `$ move_arm_with_gui` (use argument `--sim` if you don't have real robot), which allows you to move each joint with a slider.

Here is a brief example of how to set and read the joint angles of the robot using `RobotArm.move_arm_jpos` and `RobotArm.get_arm_jpos`. The API refers to joint angles as joint positions (or jpos).

```python
# joint angles in radians, from base to wrist_rotation
jpos = [0.3, 0, 0, 0, 0]
```

(continues on next page)



(continued from previous page)

```
# sends command, returns once motion stops
robot.move_arm_jpos(jpos)

# see achieved joint angles
achieved_jpos = robot.get_arm_jpos()
```

2.3 End Effector Control

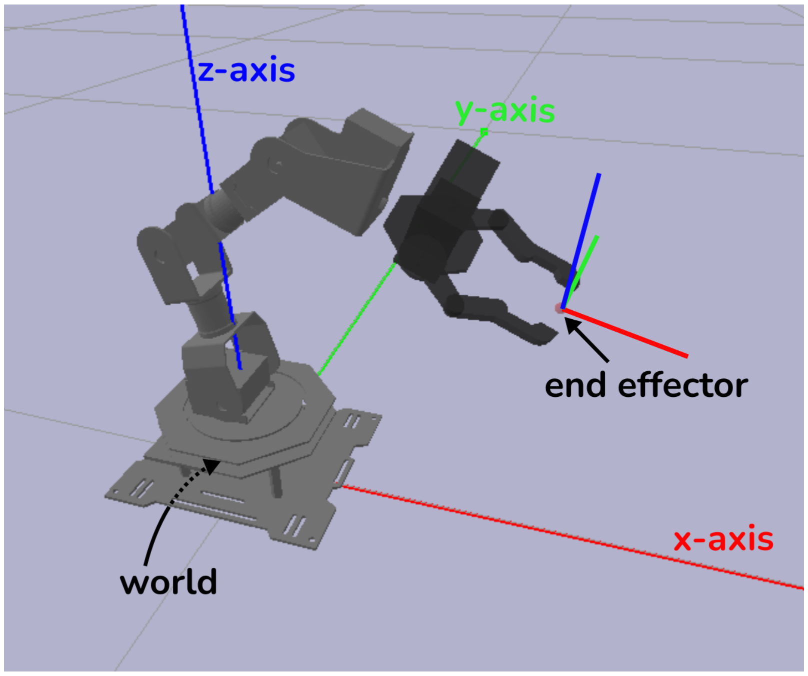
For certain tasks, like picking up an object, it is useful to command the robot to move its gripper to a specific location in the world. This is called **end effector** control. The figure below illustrates the end effector of the robot and how the position is defined. For this robot, the end effector is located at the point where the gripper fingers close (thus if an object is located at *X*, then moving end effector to *X* allows grasping the object). The position of the end effector is defined in meters, relative to the **world** (e.g. the ground underneath the center of the robot). For reference, the end effector position pictured here is about <x=0.15, y=0., z=0.08>.

Once the robot receives the desired end effector position, it performs a calculation, called Inverse Kinematics, to determine the joint angles that each motor should be set to. Below, we have a simple example showing how to move the end effector to a position and read the current end effector position.

```
# end effector position, units in meters
ee_pos = [0.2, 0.0, 0.1]

# sends command, returns once motion stops
robot.move_hand_to(ee_pos)

# see achieved end effector position
achieved_ee_pos, _ = robot.get_hand_pose()
```



2.4 Using Gripper

The gripper is controlled independently from the arm. Most of the time, it is sufficient to either fully open or fully close the gripper. However, you can also specify *how* open/closed you want the gripper to be. Examples of each of these commands are shown here:

```
# opens gripper, returns once motion stops
robot.open_gripper()

# closes_gripper, returns once motion stops
robot.close_gripper()

# for more fine-grained gripper control, use float ranging from
# 0 (fully closed) to 1 (fully opened)
robot.set_gripper_state(0.1)

# get current gripper state (ranging from 0 to 1)
gripper_state = robot.get_gripper_state()
```

Usually, it is not advised to command a movement if you know it will cause a collision. So, *do you have to anticipate an objects size when commanding how far to close the gripper when grasping?* No, instead you can use the argument `backoff` in `Robot.set_gripper_state`. Backoff determines what will happen when the gripper tries to move but encounters and object. If backoff is negative, then the gripper will apply some clamping pressure to the grasped object, without damaging the motors. By default, `RobotArm.close_gripper` sets `backoff=-0.05`. It is unlikely you will need to change this argument.

2.5 Advanced Motions

By default, the motions produced by `RobotArm.move_arm_jpos` and `Robot.move_hand_to` are linear in joint space. The speed can be controlled using the `speed` argument, which specifies the movement speed in radians per second. For safety reasons, the robot will restrict speed to be in the range of `0.1` to `4.0`. Here is an example showing two ways to adjust the movement speed:

```
# by default the speed of all motors is 1.0
robot.move_arm_jpos(jpos)

# increase speed for ALL arm joints
robot.move_arm_jpos(jpos, speed=2.0)

# restrict speed of base joint, keep others at default speed
robot.move_arm_jpos(jpos, speed=[0.5, 1, 1, 1, 1])
```

The `speed` argument is also available for `Robot.set_gripper_state`.

So far, all of the movement commands have been **safe**. Under the hood, the `RobotArm` class monitors all movements to detect and stop unanticipated collisions, so even if you tried to break something (*please don't*), you would have a hard time. For *most* applications, you should stick to these commands. However, for applications where you need complex or fast motions (for instance drawing or throwing) it is better to avoid the monitoring, which requires full stops between movements.

To run a motion without the monitoring, you can send commands directly to the servos using `RobotArm.controller.move_servos(joint_ids:  List, jpos:  List, duration:  int)`. The interface is a bit more complicated so we will show some examples uses below (see this script demonstrating how to perform cartesian control).

```
# get ordered list of joints => (base, shoulder, elbow, wrist, wristRotation, gripper)
joint_names = robot.joint_names

# move base and elbow to joint positions of 0 radians over 2000 milliseconds
robot.controller.move_servos([0, 2], [0., 0.], duration=2000)

# to get 'smooth' movements, send new command as old one is about to finish
# here we move wristRotation by increments of 0.15 radians
import time
duration_ms = 100
for i in range(10):
    robot.controller.move_servos([4], [0.15*i], duration=duration_ms)
    # sleep expects a time in seconds so divide by 1000
    time.sleep( (0.95 * duration_ms) / 1000 )

# to get multiple motors to move at different speeds, you can simply run
# commands one after the other
# here we move elbow twice the speed of wrist (e.g. half the duration)
robot.controller.move_servos([2], [0], duration=500)
robot.controller.move_servos([3], [0], duration=1000)
```

Since there is no collision detection, we recommend that you plan out motions using the simulated robot. If an unsafe motion does occur, either terminate the script or turn off the robot. There are no safeguards on the speed of movement so start with a large value of `duration` (it is in milliseconds), and then reduce as needed.

2.6 Collision Detection

Coming soon…

2.7 Care and Maintenance

- **Always** clear the area around the robot of any fragile objects before using it.
- Always turn off the robot when not in use.
- Do not run the robot for extended periods of time; if you want to stay connected while you debug, then place the robot in passive mode to prevent the motors from overheating.
- Check that there is no pinching or straining of the wires between motors. This is especially important if you are performing fast motions.
- Do not move the motors by force. If there is resistance to movement, the motors may be ON. Either turn off the robot or place it in passive mode.

CHAPTER

THREE

USING CAMERA

To use the camera, you need to install openCV. We require the contrib version, which provides support for ArUco tags. To include opencv-contrib-python in the installation, you can use the following command:

```
pip install nuro-arm[all]
```

3.1 Capturing Images

```
from nuro_arm import Camera

cam = Camera()

img = cam.get_image()
```

3.2 Calibration

If you want to associate features in the image with positions in the real world, then the camera must be calibrated.

3.3 Using ArUco Tags

ArUco Tags are visual markers that look like QR codes. Due to their unique appearance, they can be easily located in an image using a computer vision algorithm. By placing them on an object, we can determine the object's 3D position, which can be used for a robotics task.

To generate a PDF of ArUco Tags, use the *generate_aruco_tags* script. Here is an example that creates four aruco tags (ids 0 to 3) with a size of 40 millimeters.

```
generate_aruco_tags --size=40 --number=4
```

To locate ArUco Tags in an image:

```
from nuro_arm.camera.camera_utils import find_arucotags

tag_size = 0.040 # size in meters
tags = find_arucotags(img, tag_size)
```

CHAPTER

FOUR

EXAMPLES

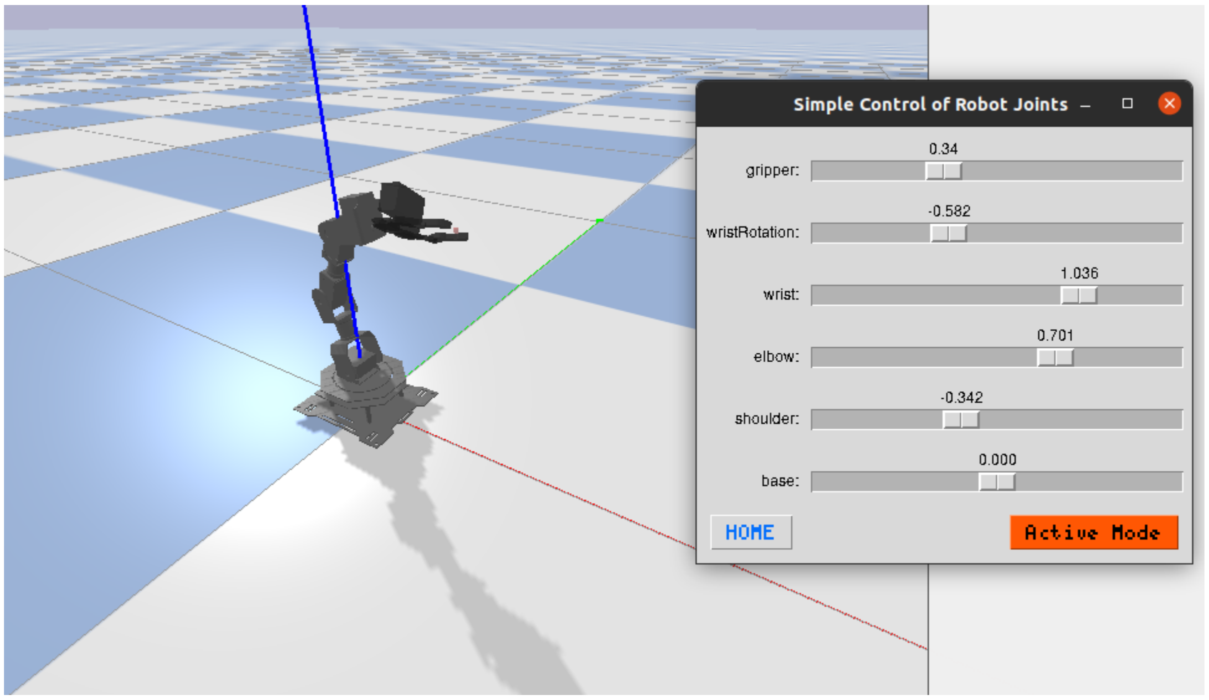
4.1 Interactive Windows

4.1.1 Joint Angle Control

The following terminal command will connect to the robot and generate a pop-up window where you can control each motor in the robot. There are two buttons at the bottom: the one on the left moves the robot to the "HOME" position, the one on the right toggles the robot from active to passive mode. When in passive mode, the window will display the current joint positions (in radians) and you are free to move the robot by hand.

```
move_arm_with_gui
```

The image below shows the simulated robot along with the pop up window. If you want to use the simulator, then use the command `move_arm_with_gui --sim`.

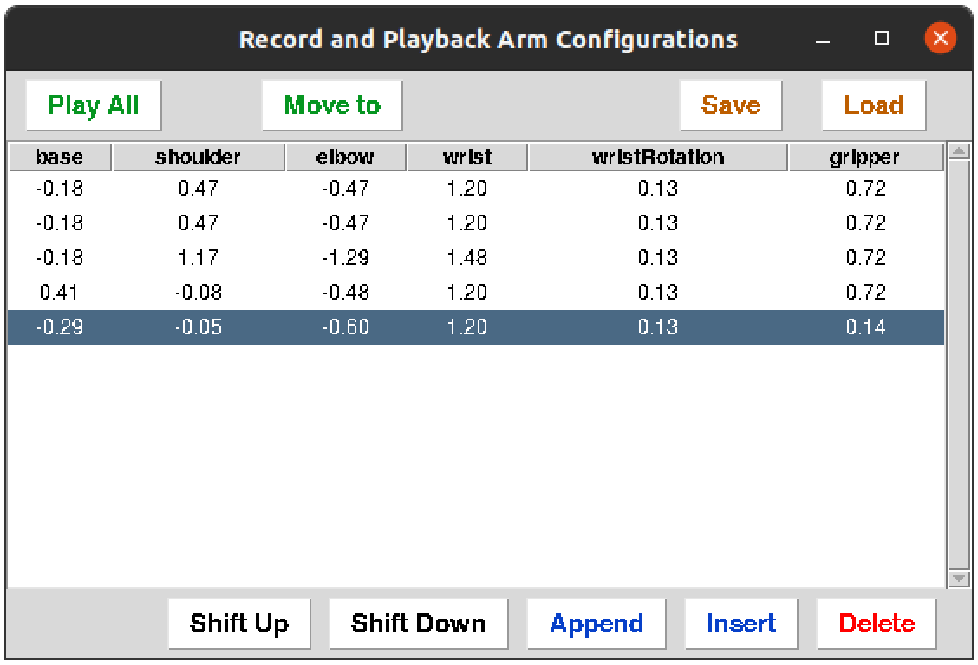


4.1.2 Recording Trajectories

For designing arm trajectories, you can run the following command in the terminal

```
record_movements
```

The robot will enter passive mode, allowing you to move it freely around. The pop up windows provide several buttons for creating and running through a sequence of joint positions. Hit “Play All” to move the robot through all rows of joint positions, or hit “Move to” to move the robot to the highlighted row. The robot will return to passive mode after it has stopped executing the motions. You can save the sequences to be loaded and modified at another time using the “Save” and “Load” buttons.



base	shoulder	elbow	wrist	wristRotation	gripper
-0.18	0.47	-0.47	1.20	0.13	0.72
-0.18	0.47	-0.47	1.20	0.13	0.72
-0.18	1.17	-1.29	1.48	0.13	0.72
0.41	-0.08	-0.48	1.20	0.13	0.72
-0.29	-0.05	-0.60	1.20	0.13	0.14

4.2 Python Scripts

4.2.1 Hardcoded Arm Movements

create a sequence of arm joint positions and use for loop to go over them

```python
from nuro_arm import RobotArm

robot = RobotArm()
jpos = [0, 0, 0, 0, 0]

for i in range(10):
    jpos[1] += i/10
    robot.move_arm_jpos(jpos)
```

4.2.2 Using Feedback from Joint Positions

Here is an example where we use the state of the gripper to determine whether an object was grasped. The program keeps attempting to close the gripper until it detects an object in the gripper (i.e. gripper could not be fully closed), at which point it drops the object off at another location.

```python
from nuro_arm import RobotArm

robot = RobotArm()
grasp_jpos = [-0.2, 0, 0.5, 0, 0]
drop_jpos = [0.2, 0, 0.5, 0, 0]

robot.open_gripper()
robot.move_arm_jpos(grasp_jpos)

while True:
    robot.close_gripper()
    gripper_state = robot.get_gripper_state()

    # if something in gripper, drop it off
    if gripper_state > 0.1:
        robot.move_arm_jpos(drop_jpos)
        robot.open_gripper()
        break

    robot.open_gripper()
```

4.2.3 Top-Down Grasping

show IK solution example

4.2.4 Nudging a Cube

show cube detection and movement

CHAPTER

FIVE

PROJECT IDEAS

Further details coming Fall 2022…

5.1 Stacking Cubes

Locate, grasp, and place cubes using aruco tags. [Advanced] Impose constraints like blue cube must be on red cube, and use planner to determine stacking order

5.2 Face Tracking

Using a camera held by the gripper, write controller that follows someone's face around as they move. [Advanced] anticipate person's motion for smoother tracking

5.3 Throwing Objects into Bin

Design motor sequence to throw object some distance. [Advanced] Parametrize motion to throw object at a specific world position.

5.4 Tactile Mapping

Use motor feedback to "map" an object. [Advanced] classify the object

5.5 Top-Down Picking

Given top down camera, use computer vision techniques to locate objects and determine grasp pose. [Advanced] use machine learning to generalize to arbitrary objects.

CHAPTER

SIX

CONTRIBUTIONS

If you use this repository for a project and want to link it on the README, please create a pull request.

We are open to adding support for other low-cost robot arms, please open an issue if you have a suggestion.

- genindex
- modindex
- search