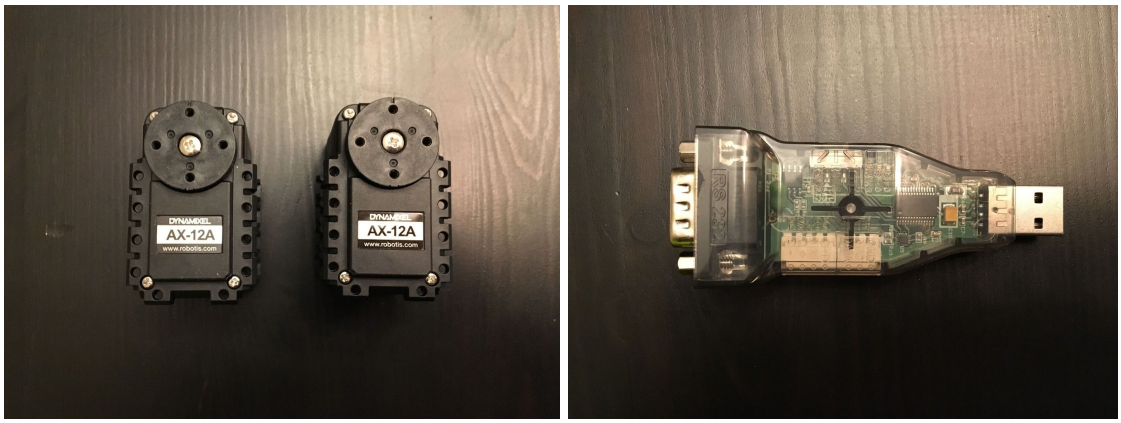
Signs of Life: Robot Module 1

Abstract

Robots can be an ideal platform for testing reinforcement learning (RL) algorithms as they move us outside of the clean environments of theory and simulation and into the messy real world. This module begins the initial setup of a robot that can be used to test out RL algorithms in a way that brings in many of the complexities of the real world, while keeping the environment constrained enough to focus on being able to test important parts of different algorithms. The first section of this report outlines the assembly of the robot and the basic physical setup. The second section covers initializing the robot and overview of what sensors and commands are available for the robot. The third section creates an initial behavior policy for the robot. While the last section logging and plotting the data created during the robot's operation. The goal at the end of this module is to have built a solid setup that will enable future real world RL experiments.

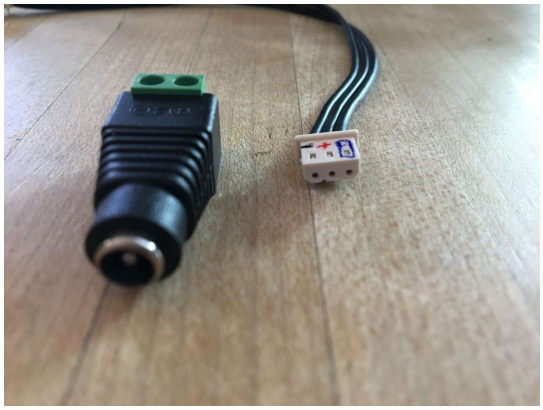
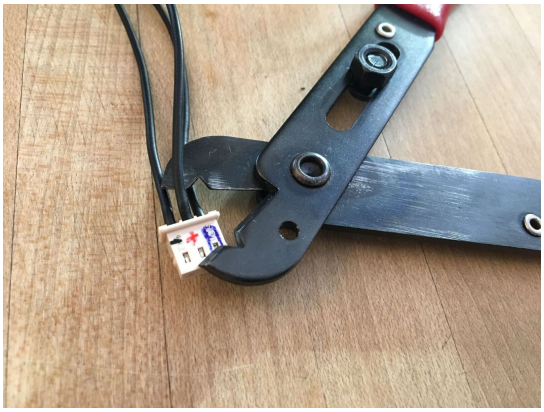
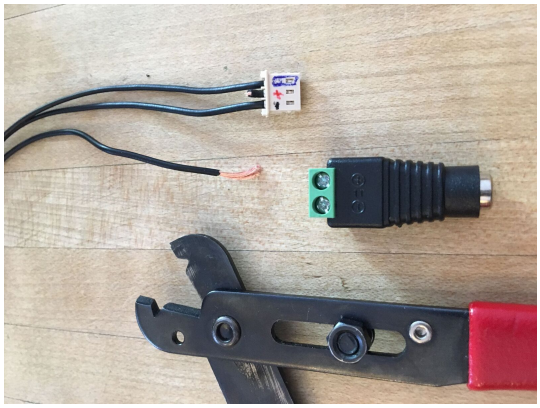
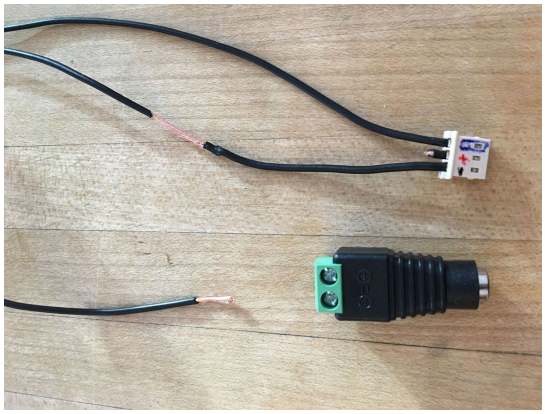
Robot Information and Assembly

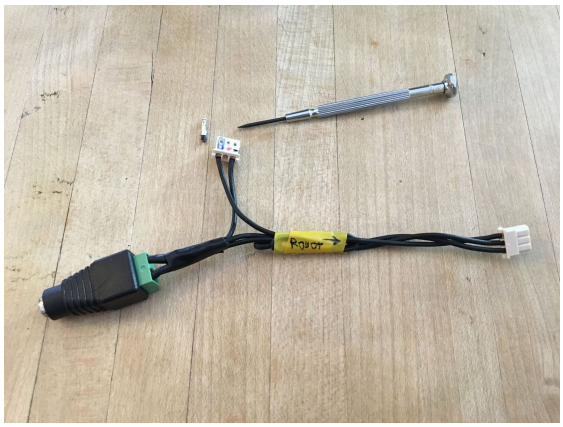
The main robot components were two Dynamixel AX-12A servos [1]. The base of one was connected to the axle of the other, and the top servo had a U-bracket attached to it. While future construction of the robot is meant to be more complicated, and allow the robot to accomplish further tasks, the goal at this point was to have the two servos connected and be able to send commands and read data from them.



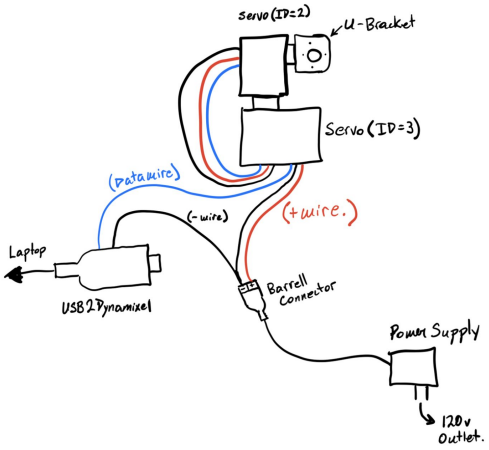
Left: 2 Dynamixel AX-12A Servos. Right: USB2Dynamixel connector. Source: 607 Dropbox.

Powering the robot, and connecting the robot to a laptop requires some assembly of the power cables. Figure 1 shows a sketch of the power setup. The basic setup requires the data wire from a USB2Dynamixel to connect to the robot, and the negative end from the USB2Dynamixel connects to the power barrel. Both the negative and positive wires from the power barrell connect into the robot along with the data cable from the USB2Dynamixel. From there a 3 wire connector connects into the second servo, and more can be chained together further.



Robot assembly (reading left to right): 1) Barrel connector and labelled wire. 2) Disconnect positive (+) wire from one end of the labelled wire. This will connect to the USB2Dynamixel. 3) Stripped positive wire for barrel connector. 4) Stripped middle of ground (-) wire which will be folded and inserted into the negative of the barrel connector. 5) Finished barrel connector, notice no exposed copper wiring, if needed tape up wires near barrel connector both for support and to make sure not copper can accidentally be touched. 6) Final wire assembly. The wire with 2 wires still connected connects to the USB2Dynamixel, while the 3 wire end connects to the robot. Notice that the stubbed ind from 2 wire end has been removed to eliminate any possible issues. Source: 607 Dropbox.



Left: Cable setup schematic. Right: Final robot construction. A BPF-WA/BU set by Robotis was used to attach the servos and add the U bracket to the top servo. [4]

Finally, a proper process needs to be followed when connecting the robot to a laptop and a power supply to keep it from getting damaged (taken from twitchy_init.py) [2]:
1) Plug in power to the robot's barrel connector
2) Plug the USB2Dynamixel (set to TTL) into the USB port of your computer
3) Plug the cable into the proximal servo
4) Plug in servo cable to USB2Dynamixel

When disassembling the robot the reverse of the above needs to be performed, first disconnecting the robot from the USB2Dynamixel, before disconnecting the power supply.

Robot Initialization

After the robot has been built and connected to a laptop it can be interacted with using a python library, lib_robotis_hack.py, released by Georgia Tech (see attached code) [1]. A few commands are necessary to finish setting up the robot, and to test to make sure it is working properly. First, the dynamixel library needs to be imported and given the location of the usb serial port that it is attached to. Next the servos need to assigned IDs. While this step is not entirely necessary, servos can come with default IDs that are the same, which can cause no servos to show up and an error to be thrown. This can be mitigated by assigning new IDs to your servos. Finally a few commands can be run to make sure that data is able to be read from the robot, as well as commands sent to the robot to move it. A walkthrough of the code is below, note that the commands are meant to be run individually in a python interpreter, not run from a single file. Adapted from twitchy_init.py [2]

```
# Import Georgia Tech Library
from lib_robotis_hack import *
# Connect USB to Serial Channel
D = USB2Dynamixel_Device(dev_name="/dev/tty.usbserial-###USB PORT
HERE###",baudrate=1000000)

# Connect first servo
s_list = find_servos(D)
# Make a servo object for this servo
s1 = Robotis_Servo(D,s_list[0])
# Set the ID of the first servo to 2
s1.write_id(2)

# Sample observations (signals) from servos
s1.read_angle()
s1.read_load()
s1.read_temperature()
s1.read_voltage()
# Send commands to servo
s1.move_angle(0.0)
s1.move_to_encoder(512)
s1.disable_torque()
s1.enable_torque()

# Plug in the next/remaining servo on your bus
s_list = find_servos(D)

# Rename the second servo
# Set this servo to ID "3"
s1 = Robotis_Servo(D,s_list[0])
s1.write_id(3)
# Rescan for the new servo names
s_list = find_servos(D)
s1 = Robotis_Servo(D,s_list[0])
s2 = Robotis_Servo(D,s_list[1])
# Read the angle of servo 1
s1.read_angle()
# Read the load of Servo 1
s1.read_load()
# Read the angle of servo 2
s2.read_angle()
# Read the load of servo 2
s2.read_load()
# Move both servos to angle 0.0
s1.move_angle(0.0); s2.move_angle(0.0)
# Move both servos to angle 0.5
s1.move_angle(0.5); s2.move_angle(0.5)
```

Robot Control

A basic behavior policy was implemented for the robot with one servo moving between and angle of -1.0 and 1.0, and the second servo moving between -0.75 and 0.75. These numbers were chosen both to test the range of the robot, but also to make sure they were running different commands at different times to help debug any blocking issues. Neither of the operations were blocking, meaning that sending a command to a servo did not stop commands from being run on the other servo, and both servos were able to have data read from them during operation.

```
for x in range(1, 1000):
    if not s1.is_moving():
        target1 = -target1
        s1.move_angle(target1, blocking=False)
    if not s2.is_moving():
        target2 = -target2
        s2.move_angle(target2, blocking=False)
```

All code for this module was kept in one file with all of the behavior and logging code kept in one loop. While breaking it out into multiple files may be best for the future it made things more clear to see what was happening in one loop while doing this initial setup. [3]

Dealing with Data

Graphing data

Graphing was done using Visdom [5], a real time data library that uses Plotly [6] for charts, released by Facebook. Following installation on the Visdom page [5] (for most systems pip install visdom will work), you start a visdom server using python -m visdom.server. After this you are able to send graphing commands to Visdom, which you can see the results of by navigating to http://localhost:8097 in a web browser. A basic line graph is given by:

```
win = vis.line(
    X=np.column_stack((np.array([0]), np.array([0]), np.array([0]),
                       np.array([0]))),
    Y=np.column_stack((np.array([0]), np.array([0]), np.array([0]),
                       np.array([0]))),
    opts=dict(
        showlegend=True,
        width=300,
        height=300,
        xlabel='Time',
```

```
        ylabel='Reading',
        title='Servo 1',
        marginleft=10,
        marginright=10,
        marginbottom=30,
        margintop=10,
        legend=['angle', 'load', 'voltage', 'temp'],
        ),
    )
```

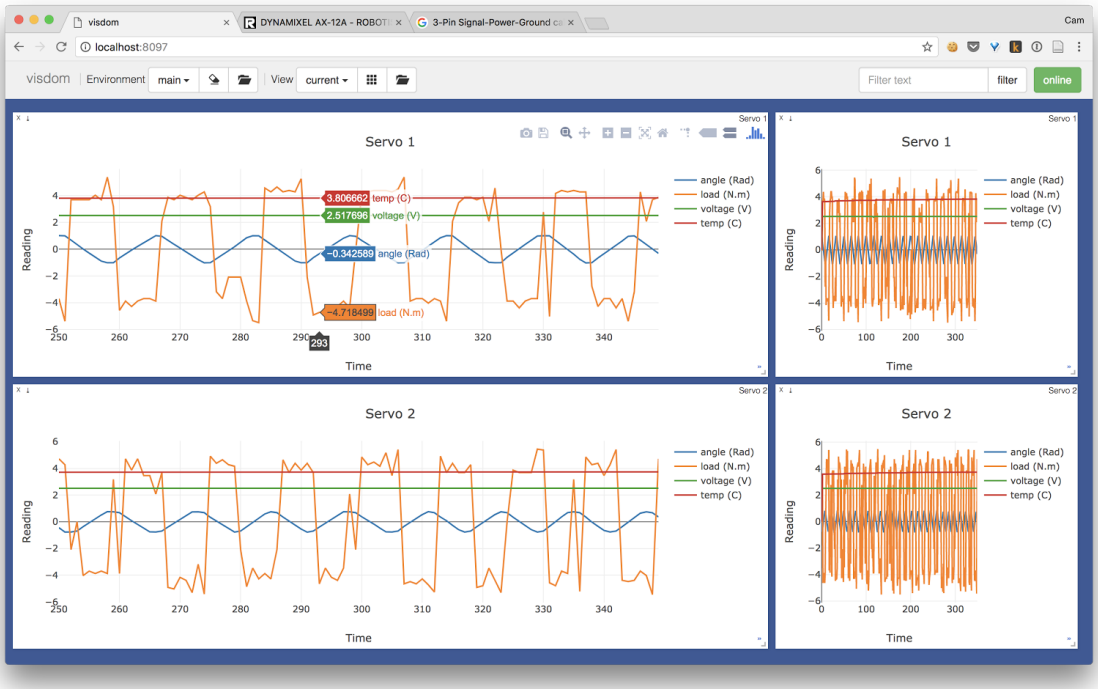
With new data added to the plot on each time step using:

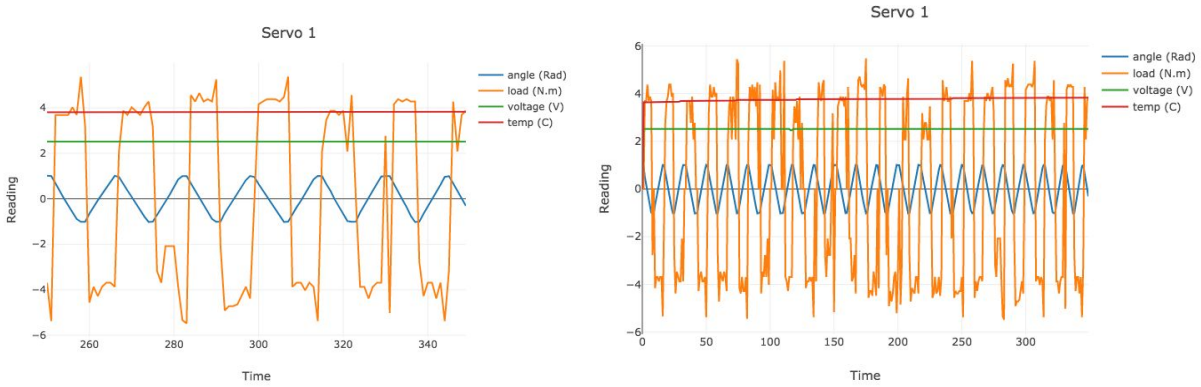
```
vis.line(
    X=np.column_stack((np.array([x]), np.array([x]), np.array([x]),
                       np.array([x]))),
    Y=np.column_stack((np.array([angle1]), np.array([load1]), np.array([volt1]),
                       np.array([temp1]))),
    win=win,
    update='append'
    )
```

This method will graph all data during operation. A more granular view on the operation of the robot may be desired. To do this we want to view only a window of operational data. That is accomplished by using the python collections library to create a queue of data to graph, and changing our update to ‘replace’, and using the windowed data. Code can be seen below.

```
windowed_angle1 = collections.deque(100*[0], 100)
windowed_load1 = collections.deque(100*[0], 100)
windowed_volt1 = collections.deque(100*[0], 100)
windowed_temp1 = collections.deque(100*[0], 100)
windowed_angle1.append(angle1)
windowed_load1.append(load1)
windowed_volt1.append(volt1)
windowed_temp1.append(temp1)
vis.line(
        X=np.column_stack((x_window, x_window, x_window, x_window)),
        Y=np.column_stack((windowed_angle1, windowed_load1,
                           windowed_volt1, windowed_temp1)),
        win=win3,
        update='replace',
        opts=dict(
          legend=['angle (Rad)', 'load (N.m)', 'voltage (V)', 'temp (C)'],
        )
    )
```

This produces a dashboard seen below.





Top: Dashboard of real time data. Left: Windowed data during operation. Right: Data after a minute of operation. Note that all the data except for the angle are log scaled to keep graphs to a more manageable size.

Logging data

While Visdom keeps a log of the data it plots, we also want to have data saved in an easy to access format for offline analysis. This is done by using numpy's savetxt function:

```
def log_data(data, filename):
    with open(filename, 'ab') as f:
        np.savetxt(f, data, delimiter=",", newline=" ",
                   footer="\n", comments="")
```

With data then easily saved on each step:

```
log_data([x, target1, angle1, load1, volt1, temp1], "servo1.dat")
```

Data can then be read back in and further analyzed. It can also be sent to Visdom to be analyzed.

```
def plot_offline_data(filename):
    data =  np.loadtxt(filename)
    columns = [data[:,c] for c in range(1, len(data[0]))]
    win = vis.line(
        Y=np.column_stack(columns),
    )
```

Conclusion

This project outlines the initialization of robot that can be used to test reinforcement learning algorithms. It outlines the setup and initialization of a basic robot using two Dynamixel AX-12A servos. Future work will include further adding to the structure of the robot, as well as beginning to implement learning algorithms to make predictions about the robot.

A video of the robot in action with live graphing can be seen at:
https://www.youtube.com/watch?v=hU5n8kF8sCA&feature=youtu.be&app=desktop

Code can be found at: https://github.com/camlinke/607/tree/master/module1

The main experimental file can be found at:
https://github.com/camlinke/607/blob/master/module1/module1.py

References

1 https://github.com/camlinke/607/tree/master/module1
2 Twitchy Init. 607 Dropbox Folder
3 https://github.com/camlinke/607/blob/master/module1/module1.py
4 http://www.robotis.us/ax-series/
5 https://github.com/facebookresearch/visdom
6 https://plot.ly/