# Monash Deep Neuron: Motorola Project:

## Ursim Robot arm simulation for mind control pipeline.

Last updated 10/Jul/25

---

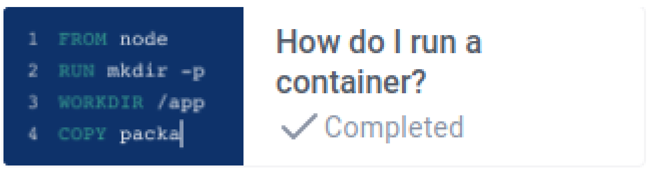
The following project allows the user to control a simulated robot arm using a python script that takes in a json file of streaming commands.
This is advantageous for prototyping uses and is suited at the end of the pipeline.

## Step 1: Download + understand Docker

**1.1**: Get docker from here:
[Docker Desktop: The #1 Containerization Tool for Developers]

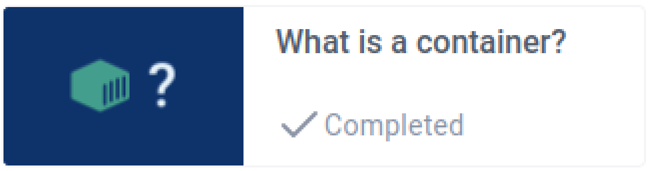
Docker contains **containers**
Images can exist in containers
Images are made from dockerfile + sample application

**1.2:** Do these in the learning centre:
You'll learn how to make the container using code and dockerfile:



```
FROM node
RUN mkdir -p
WORKDIR /app
COPY packa
```

How do I run a container?
✓ Completed



What is a container?
✓ Completed

These will teach you how to pull and run a container from a code and thus image.
It will not teach you how to access files outside the container, nor how to access the container's results persisting outside the container lifetime. There are other lessons you may watch for that but are not necessary.
However, the docker app just needs to be open for the rest of the steps.

# Step 2: Prepare the Ursim container

## 2.1 - Get Ursim from here

Make a new directory first, Lets say ursim
Inside we pull the image from docker of the simulator.

```
docker pull universalrobots/ursim_e-series:latest
```

## 2.2 - Prepare the python venv

We make a virtual environment in python for clarity to store and containerise all the necessary packages in line with the docker container, making everything rather neat and tidy.
We also install ur_rtde package here, which allows us to control the simulated robot.
(This venv is called ur_venv)

```
python3 -m venv ur_venv
source ./ur_venv/bin/activate
pip install ur_rtde
```

Remember a venv can be closed via *deactivate*

## 2.3 - Placing necessary executables

Now we will also be putting two files, they should come attached with this documentation
ur_example.py, which is our executable python file,
And startDocker.sh, which is our executable docker container maker from the image we made.

These all go under the main folder hence:

```
>>Ursim
        >> ur_venv
        >> startDocker.sh
        >> ur_example.py
```

## 2.4 - You can start your docker container now.

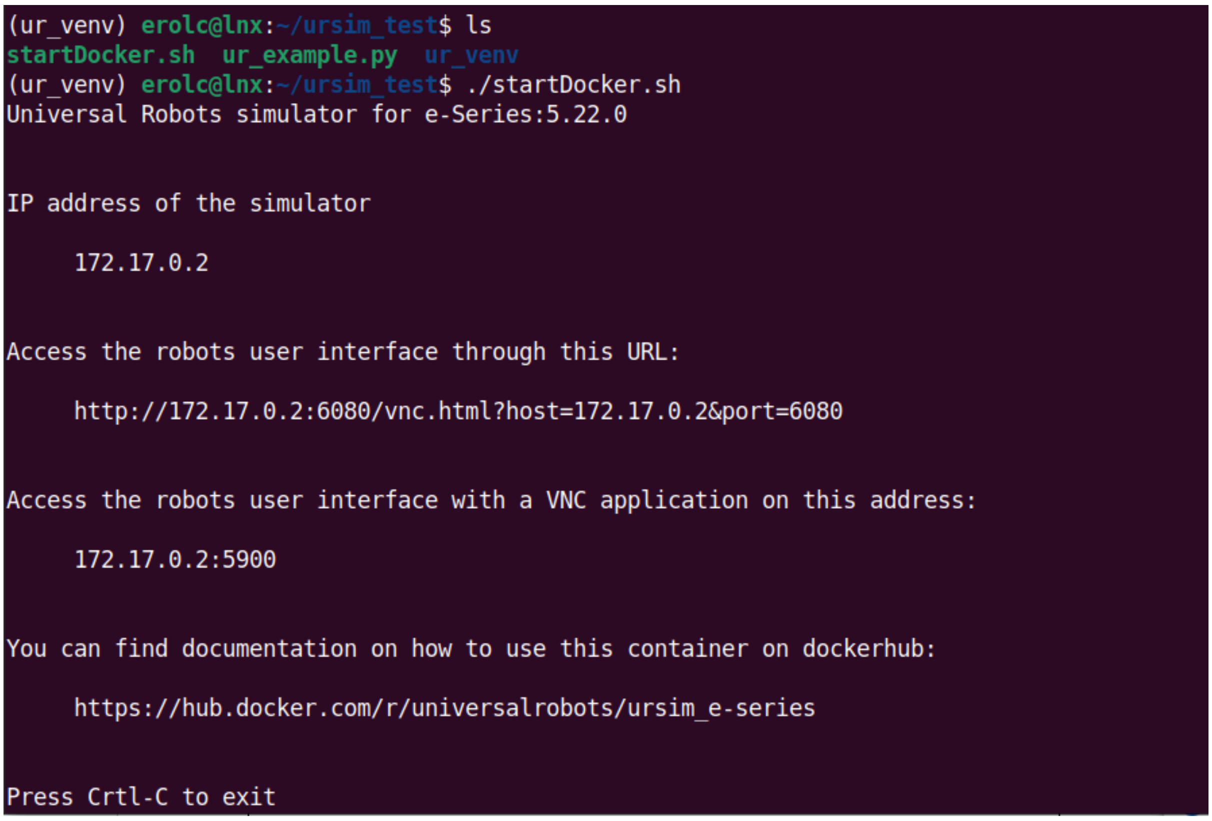
Remember to still be in the folder; Ursim.
We use the docker executable for this, recall in some systems, you may need to provide permissions to do this:
Eg. on Linux: chmod +x startDocker.py

```
./startDocker.sh
```

If you are curious what is in startDocker commands: it is:

```
docker run --rm -it \
  -p 5900:5900 \
  -p 6080:6080 \
  -p 29999:29999 \
  -p 30001-30004:30001-30004 \
  -e ROBOT_MODEL=UR5E \
  -e URSIM_ROBOT_MODE=SIMULATION \
  --name ursim_e_series \
  universalrobots/ursim_e-series
```

Should you be successful, you should see the terminal print the following:



```
(ur_venv) erolc@lnx:~/ursim_test$ ls
startDocker.sh  ur_example.py  ur_venv
(ur_venv) erolc@lnx:~/ursim_test$ ./startDocker.sh
Universal Robots simulator for e-Series:5.22.0


IP address of the simulator

     172.17.0.2


Access the robots user interface through this URL:

     http://172.17.0.2:6080/vnc.html?host=172.17.0.2&port=6080


Access the robots user interface with a VNC application on this address:

     172.17.0.2:5900


You can find documentation on how to use this container on dockerhub:

     https://hub.docker.com/r/universalrobots/ursim_e-series


Press Crtl-C to exit
```

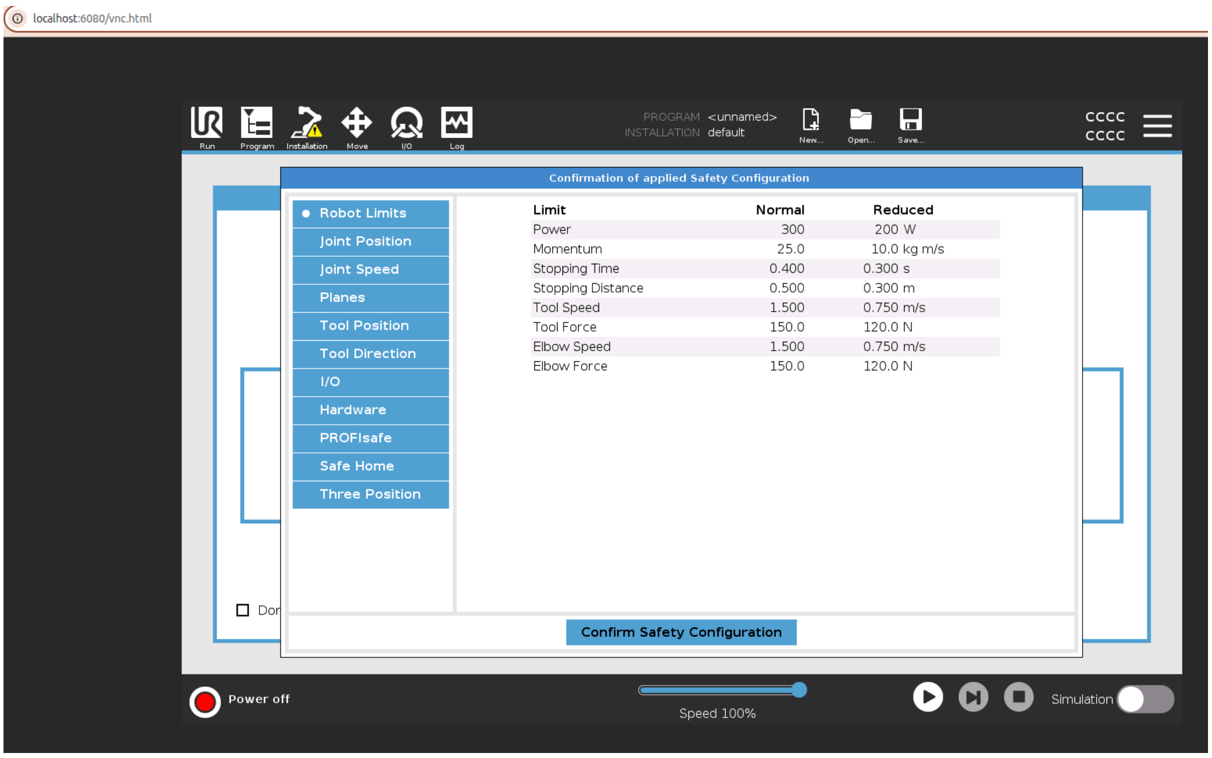
## Step 3: Prepare the Ursim container

### 3.1 - Open the simulator:

Simply click on this link:

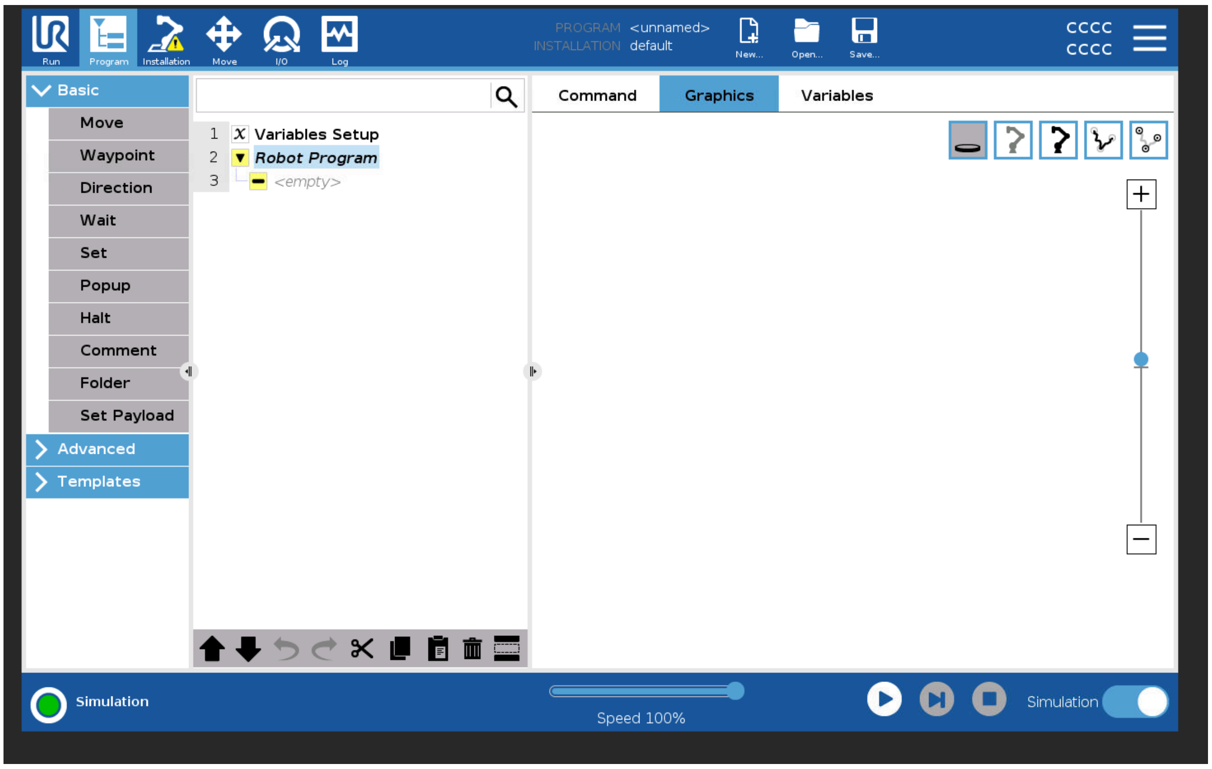
```
http://localhost:6080/vnc.html
```

Which should take you to here

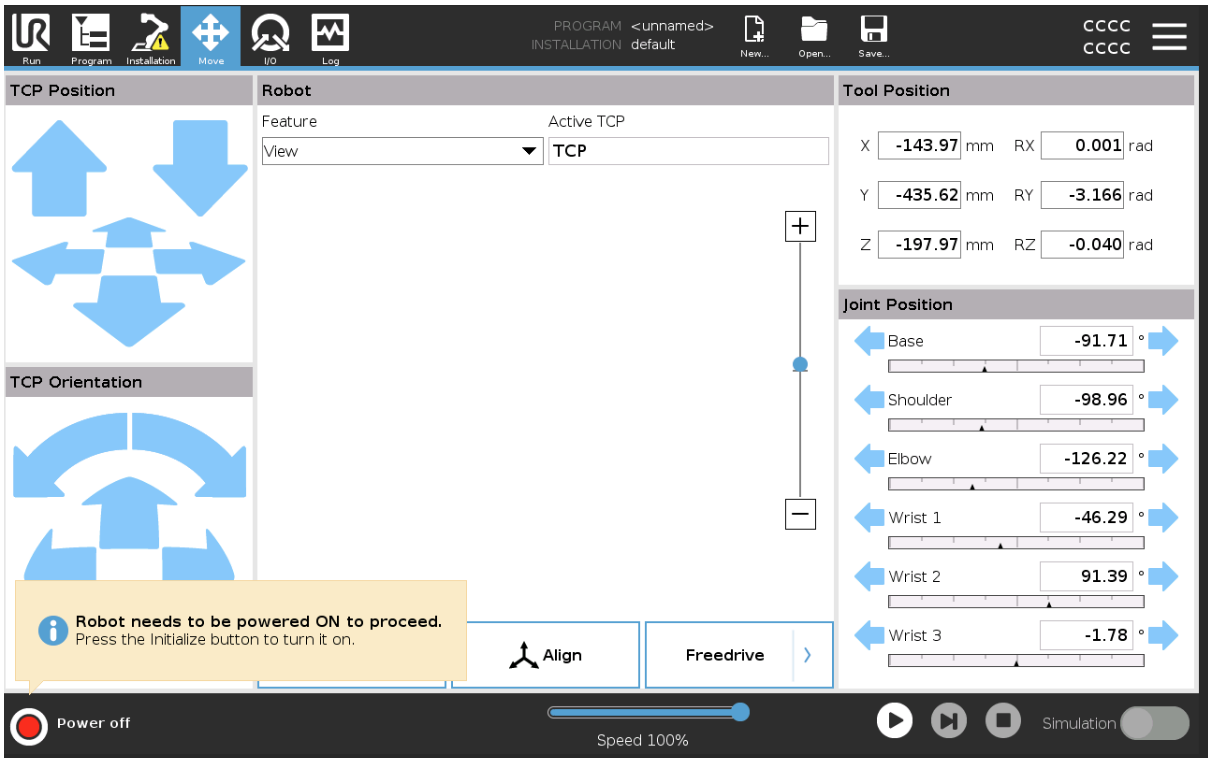


| Limit | Normal | Reduced |
|---|---|---|
| Power | 300 | 200 W |
| Momentum | 25.0 | 10.0 kg m/s |
| Stopping Time | 0.400 | 0.300 s |
| Stopping Distance | 0.500 | 0.300 m |
| Tool Speed | 1.500 | 0.750 m/s |
| Tool Force | 150.0 | 120.0 N |
| Elbow Speed | 1.500 | 0.750 m/s |
| Elbow Force | 150.0 | 120.0 N |

Press [Confirm Safety Configuration]
And then at the bottom right corner, turn ON [Simulation]
Then go to Program → Graphics: This is where the robot simulation will take place. Notice, however, that there is no robot showing:



So to activate the robot arm we need a second step: Go to Move on the top left corner.



Robot needs to be powered ON to proceed.
Press the Initialize button to turn it on.

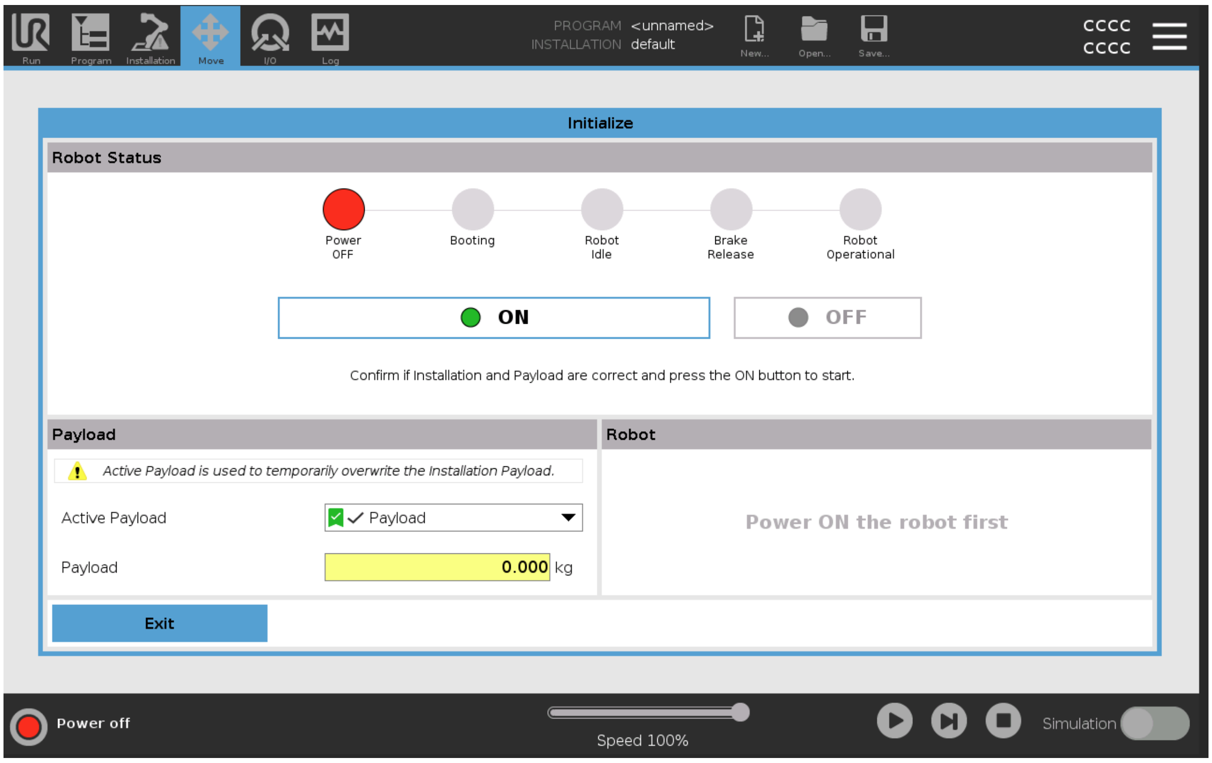
Tool Position

| | | | | | |
|---|---|---|---|---|---|
| X | -143.97 mm | RX | 0.001 rad |
| Y | -435.62 mm | RY | -3.166 rad |
| Z | -197.97 mm | RZ | -0.040 rad |

Joint Position

| | |
|---|---|
| Base | -91.71 ° |
| Shoulder | -98.96 ° |
| Elbow | -126.22 ° |
| Wrist 1 | -46.29 ° |
| Wrist 2 | 91.39 ° |
| Wrist 3 | -1.78 ° |

When you go in this menu, you'll notice that there is still no robot in the middle. But you get a suggestion to power ON the robot via the bottom left button: Press that.



Confirm if Installation and Payload are correct and press the ON button to start.

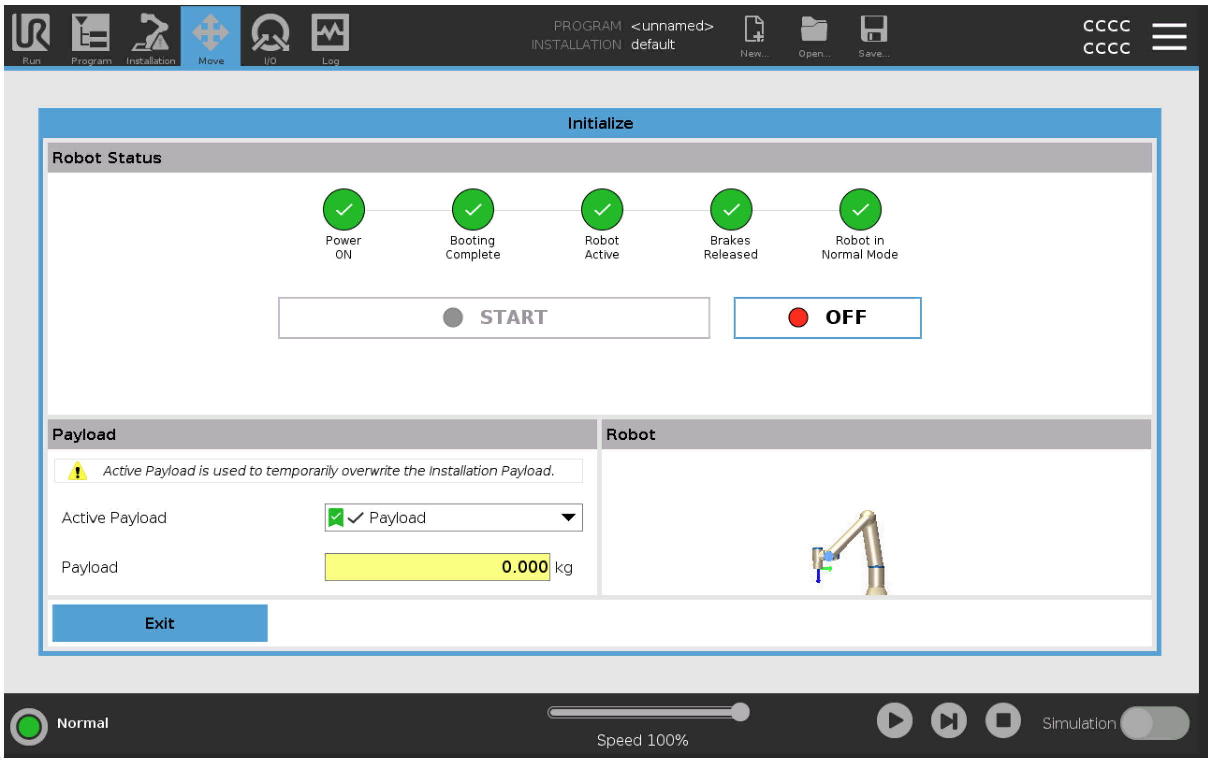
Active Payload is used to temporarily overwrite the Installation Payload.

Active Payload: ✓ Payload

Payload: 0.000 kg

Power ON the robot first

Press ON, then press START to take the robot out of the safety brake and allow all the 5 circles to be green; this means the robot is operational and ready

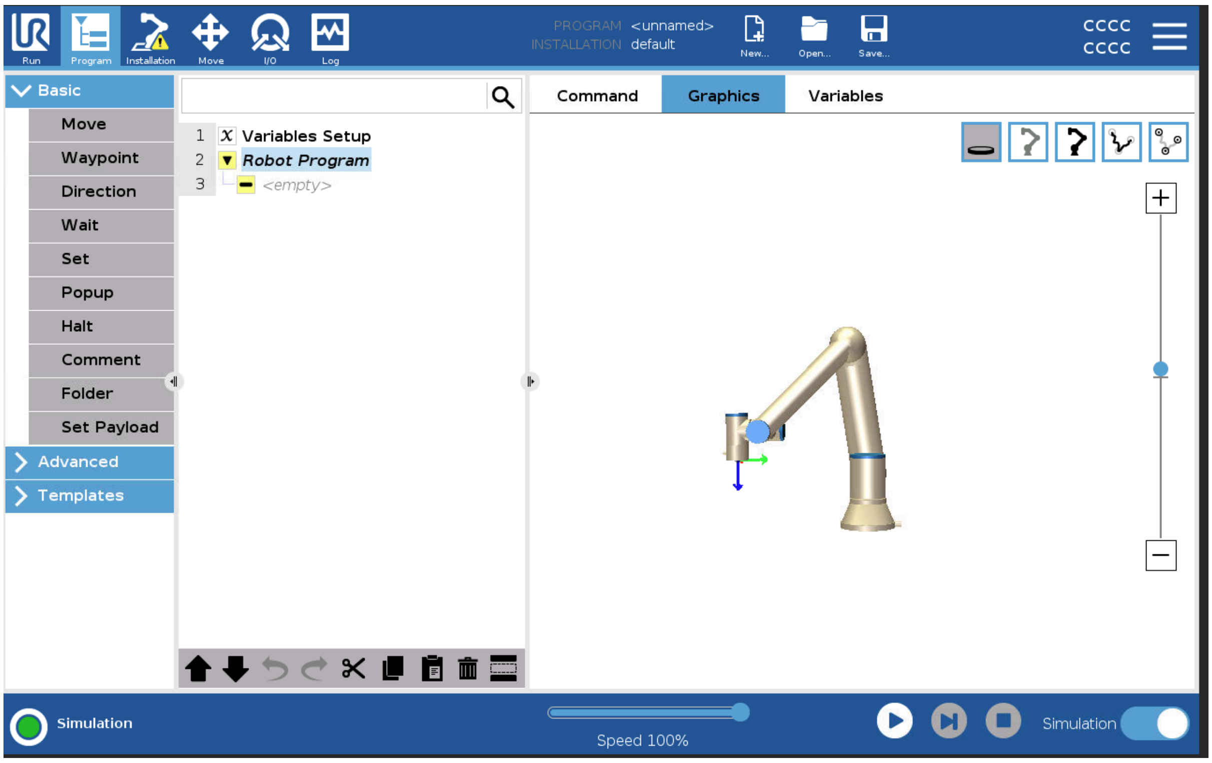


Active Payload is used to temporarily overwrite the Installation Payload.

Active Payload: ✓ Payload

Payload: 0.000 kg

Here, now just press EXIT at the bottom left corner, and then go back to PROGRAM -> Graphics, you should see the robot like the following:



```
1 Variables Setup
2 Robot Program
3   <empty>
```

(Just double check that Simulation at the bottom right corner is still turned ON)

## Step 4: Run the test python script:

### 4.1 - Open a new terminal tab in the same directory

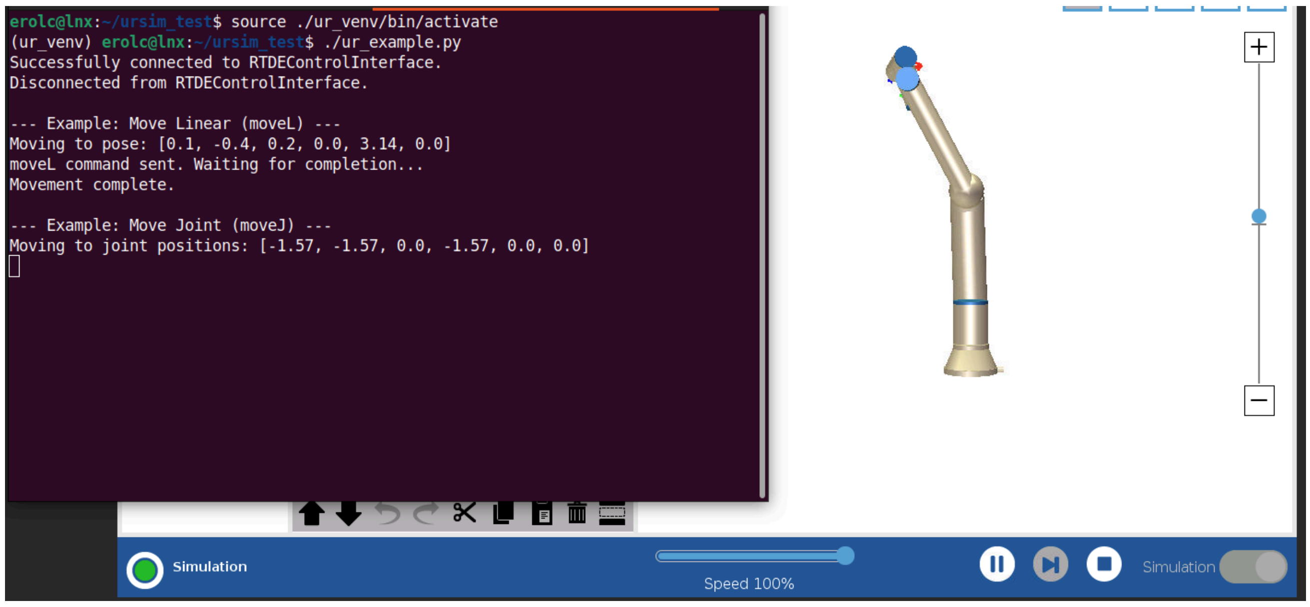
The python script is another node that runs concurrently with this docker container, it should ideally run continuously and thus can take streaming data such as from a json file.
We do this by opening a new terminal in the same directory.
Remember to activate the venv!

>> ursim

```
source ./ur_venv/bin/activate
```

### 4.2 - Run the python file executable

```
./ur_example.py
```



```
erolc@lnx:~/ursim_test$ source ./ur_venv/bin/activate
(ur_venv) erolc@lnx:~/ursim_test$ ./ur_example.py
Successfully connected to RTDEControlInterface.
Disconnected from RTDEControlInterface.

--- Example: Move Linear (moveL) ---
Moving to pose: [0.1, -0.4, 0.2, 0.0, 3.14, 0.0]
moveL command sent. Waiting for completion...
Movement complete.

--- Example: Move Joint (moveJ) ---
Moving to joint positions: [-1.57, -1.57, 0.0, -1.57, 0.0, 0.0]
```

Opening the website again while this is running, you should see the the robot do a series of movement tests.

**Ur_example.py** is a testing document to validate that the simulation is working as expected. And simply diagnoses the operational state of this simulation system.

If the system has worked correctly, we can move on with the more interesting python file, **ur_synchronous.py** which will actuate commands based on the stream of json files entering.

## Step 5: Operate the arm using json files:

### 5.1 - Place the operation files

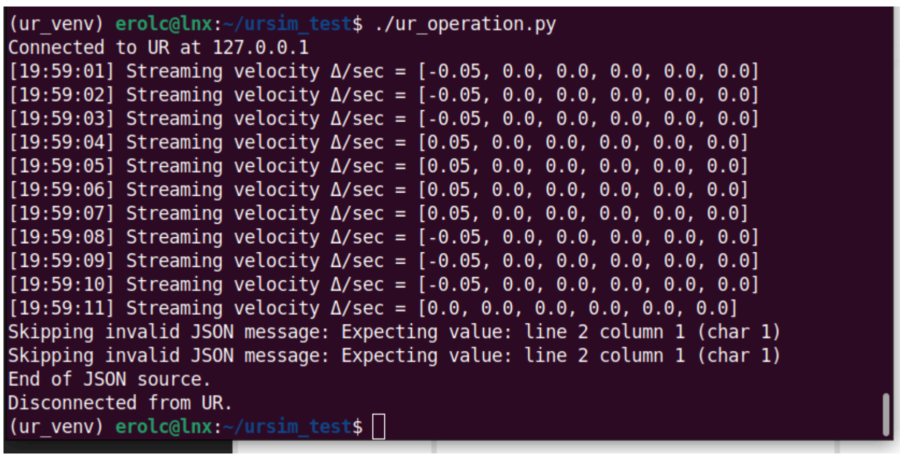
Attached you will also find a python file called
**ur_synchronous.py**, using that, place under the ur_venv directory, same as all the other files we've put so far.
Additionally, there is an example commands input jsonl file for use to test this code called **synchronous_deltas.jsonl**

Your directory should look like this now:
>> ursim_test
    >> ur_example.py
    >> ur_synchronous.py
    >> synchronous_deltas.jsonl
    >> ur_venv

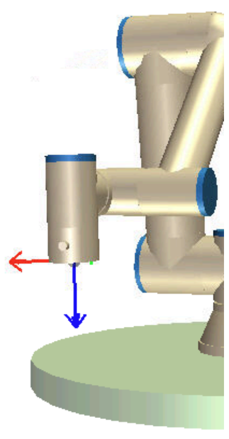
If you investigate the json file, you'll find it simply moves the arm one direction and then back, feel free to modify it to your preference.

### 5.2 - Operation

```
./ur_synchronous.py
```



```
(ur_venv) erolc@lnx:~/ursim_test$ ./ur_operation.py
Connected to UR at 127.0.0.1
[19:59:01] Streaming velocity Δ/sec = [-0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:02] Streaming velocity Δ/sec = [-0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:03] Streaming velocity Δ/sec = [-0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:04] Streaming velocity Δ/sec = [0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:05] Streaming velocity Δ/sec = [0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:06] Streaming velocity Δ/sec = [0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:07] Streaming velocity Δ/sec = [0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:08] Streaming velocity Δ/sec = [-0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:09] Streaming velocity Δ/sec = [-0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:10] Streaming velocity Δ/sec = [-0.05, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:59:11] Streaming velocity Δ/sec = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Skipping invalid JSON message: Expecting value: line 2 column 1 (char 1)
Skipping invalid JSON message: Expecting value: line 2 column 1 (char 1)
End of JSON source.
Disconnected from UR.
(ur_venv) erolc@lnx:~/ursim_test$
```



You should find the arm moving left and then right and then left again.

### 5.3 - How to modify the json commands file?

How does the jsonl work?
Jsonl is a json file where each line represents an appendable, separate instance, in this case it is an instruction.

Here is synchronous_deltas.jsonl

```
{"dx": -0.05, "dy": 0.00, "dz": 0.00}
{"dx": -0.05, "dy": 0.00, "dz": 0.00}
{"dx": -0.05, "dy": 0.00, "dz": 0.00}
{"dx": 0.05, "dy": 0.00, "dz": 0.00}
{"dx": 0.05, "dy": 0.00, "dz": 0.00}
{"dx": 0.05, "dy": 0.00, "dz": 0.00}
{"dx": 0.05, "dy": 0.00, "dz": 0.00}
{"dx": -0.05, "dy": 0.00, "dz": 0.00}
{"dx": -0.05, "dy": 0.00, "dz": 0.00}
{"dx": -0.05, "dy": 0.00, "dz": 0.00}
{"dx": 0.00, "dy": 0.00, "dz": 0.00}
```

Note how it moves the arm left, then right and then left again (dx represents speed and it applies once every second)

A valid extra 3 commands is also:

```
{"dx": 0.00, "dy": 0.00, "dz": 0.00 "drx": 0.00, "dry": 0.00, "drz":
0.00}
```

Where drx dry and drz represent *rotation* of the planes where the end of the arm is located relative to its current coordinate orientation.

The python node will apply once every second to apply the next line in the jsonl file, if there are no lines left, it will abort. However, jsonl files can have new lines dynamically added to them. Thus if a software wishes to add commands in real time, it need only to add a command once a second to achieve this.

Should you wish to change the frequency of these pre prepared commands, there are some flags you can call with the function ur_synchronous.py.

```
    parser = argparse.ArgumentParser(
        description="Stream JSON delta movements once per second and
move the UR robot accordingly."
    )
    parser.add_argument(
        "--robot-ip", default="127.0.0.1",
        help="IP address of the UR robot or simulator"
    )
    parser.add_argument(
        "--json-source", default="synchronous_deltas.jsonl",
        help="Path to JSON-line file of {'dx','dy','dz',…} messages (use '-' for stdin)"
    )
    parser.add_argument(
        "--json-log", default="synchronous_log.jsonl",
        help="Path to append timestamped JSON log"
    )
    parser.add_argument(
        "--speed", type=float, default=0.2,
        help="Cartesian speed (unused with speedL, placeholder)"
    )
    parser.add_argument(
        "--acceleration", type=float, default=0.5,
        help="Acceleration for speedL (m/s²)"
    )
    parser.add_argument(
        "--responsiveness", type=float, default=1.0,
        help="How often each command line in json file takes effect (seconds)"
    )
```

The jsonl file that is logged, sourced from, robot ip can all be selected via arguments. (However the defaults are conveniently preselected for you)

The acceleration argument changes the relative speed of the movements should you wish an overall slower or faster robot.
The responsiveness command refers to the frequency of commands the robot takes from the jsonl file, this is where the 1 second comes from, but can be customised to meet a more dynamic and lower latency robot.

To apply these arguments, simply call in the terminal such as this example:

```
./ur_synchronous.py --responsiveness 2
```

In this case the robot commands only read the jsonl new line once every 2 seconds now.

## Step 6: Having a streaming pipeline

### 6.1: - Place the new operation files

Whilst this synchronous file set can perform robot commands, it is still a step from being real time responsive. As it is unlikely for commands to exactly match once a second (or arbitrary time step); perfect synchronization is difficult and unnecessary.

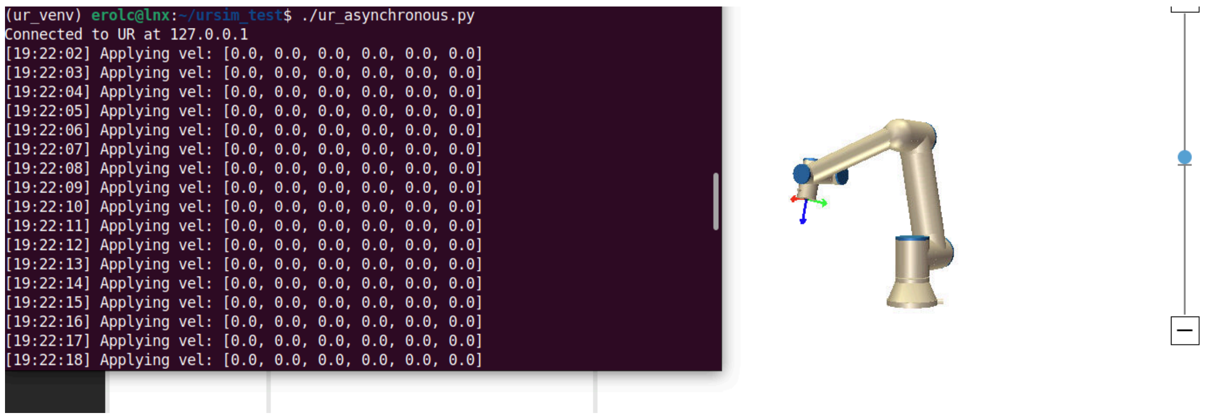
This issue is solved with a different implementation, one of which is for users who wish for a true streaming implementation of jsonl commands to the robot arm.

Attached you will also find a python file called
**ur_asynchronous.py**, using that, place under the ur_venv directory, same as all the other files we've put so far.
Additionally, there is an example commands input jsonl file for use to test this code called **asynchronous_deltas.jsonl**

Your directory should look like this now:
>> ursim_test
- >> ur_example.py
- >> ur_synchronous.py
- >> synchronous_deltas.jsonl
- >> ur_asynchronous.py
- >> asynchronous_deltas.jsonl
- >> ur_venv

## 6.2 - Operation

```
./ur_asynchronous.py
```



```
(ur_venv) erolc@lnx:~/ursim_test$ ./ur_asynchronous.py
Connected to UR at 127.0.0.1
[19:22:02] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:03] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:04] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:05] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:06] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:07] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:08] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:09] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:10] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:11] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:12] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:13] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:14] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:15] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:16] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:17] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[19:22:18] Applying vel: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

What you will find is that, there will be no movement, but rather, a constant reading thats streamed to the robot without stopping. This streaming node is always on until it is stopped via ctrl+c.

If you'd like to see the robot move, append a new line to the jsonl file.

The asynchronous node will read the last line in the file every second, in essence applying the most recent velocity to the robot simulation.

This update can be sped up with the responsiveness argument as before: refer to the asynchronous arguments for any other requests:

```
        description='Read tail of JSONL and apply delta velocities
each second'
    )
    parser.add_argument('--robot-ip', default='127.0.0.1', help='UR
IP')
    parser.add_argument('--json-file',
default='asynchronous_deltas.jsonl', help='source jsonl file')
    parser.add_argument('--acceleration', type=float, default=0.5,
help='Acceleration')
    parser.add_argument('--responsiveness', type=float, default = 1.0,
help='responsiveness')
```

With this done, as long as your docker container is running for the robot, you have a working simulation.
And as long as you have the ur_asynchronous.py running,
Whatever velocity information is in the final line of the asynchronous_deltas.jsonl file, it will be streamed to the robot arm.

## Step 7: BONUS: How to apply pipeline to a real UR arm.

TBA