Poppy project Documentation

Release 1.1.0

INRIA

October 09, 2015

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Getting Started

1.1.1 Introducing the Poppy project

Poppy is an open-source platform for the creation, use and sharing of interactive 3D printed robots. It gathers an interdisciplinary community of beginners and experts, scientists, educators, developers and artists, that all share a vision: robots are powerful tools to learn and be creative.

The Poppy community develops robotic creations that are easy to build, customise, deploy, and share. We promote open-source by sharing hardware, software, and web tools.

The Poppy project has been originally released by the INRIA Flowers.

This project aims to have a robotic open-source and open-hardware kit, for the conception and programming of robotic creatures. This kit also contains an integrated software environment, easily installable, dedicated to the use of the robot, and associated with a web platform enabling the community to share experiences and to contribute to the improvement of the platform.

Recently, Poppy has been used as a "robotic tool". It can be a great educational and motivational tool to learn engineering and digital sciences.

The Poppy project also consists of a very active and diverse community. People of different horizons collaborate to improve the project ; they add features to the core libraries, release advanced behaviors, create pedagogical content or even new robots. To facilitate these exchanges two supports are available:

- the forum "poppy-project" for help, dialogue and to share ideas
- github to deposit your contributions

All sources of the Poppy project (software and hardware) are available on our Github.

The Poppy Creatures

Poppy creatures are open-source and open-hardware/software robots, available to download and modify (Creative Commons License, BY-SA). They are created with the same characteristics.

All Poppy Creatures are:

- made from pieces of printable 3D and low cost engines,
- using an embedded board for control (a Raspberry Pi-2 or Odroid for older versions),
- based on a Python library, pypot, allowing to control Dynamixel servomotors in an easy way,
- have a simulator available (based on vrep),
- have a visual programming language (Snap! a variation of Scratch),
- and textual language Python.

Poppy robots (real or in the free simulator) are programmable through a "Rest API", which enables the control with any programming language through simple http requests (connection for Matlab and Processing already exists).

They can be used as it is, or hacked to explore the shape of novel legs, arms, hands...

You can get a full Poppy Creature robot from one of Poppy's official resellers:

- Génération Robots, or you can get all the parts yourself.

The 3 main Poppy Creatures:

Poppy Humanoid

It's a 25-degree of freedom humanoid robot with a fully actuated torso. Used for education, research (walk, human-robot interactions) or art (dance, performances). From a single arm to the complete humanoid, this platform is actively used in labs, engineering schools, FabLabs, and artistic projects.

You can get all the parts yourself following the Bill of Material. The 3D models for the parts can be found here.

After assembling your robot, try the discover quickstart, then have a look at the poppy_humanoid library.

Poppy Torso

It's an upper part of Poppy Humanoid (13 degrees of freedom). Poppy Torso is more affordable than the full kit (Poppy Humanoid), which makes it especially suitable for uses in an educational, associative and makers context. Poppy Torso is an ideal medium to learn science, technology, engineering and mathematics (STEM).

You can get all the parts yourself following the Bill of Material. The 3D models for the parts can be found here (they are the same as Poppy Humanoid, simply remove the legs and add the support).

After assembling your robot, try the discover quickstart, then have a look at the poppy_torso library.

Poppy Ergo Jr

The Poppy Ergo Jr robot is a small and low cost 6-degree-of-freedom robot arm. It is made of 6 cheap motors (XL-320 Dynamixel servos) with 3D-printed parts (based on OpenScad and assembled with OLLO rivets) simple to design. At the end of his arm, you can choose among several ends: a lamp, a gripper hand,... And you can change the end easily with easy to handle rivets. The 3D parts were made so they can be easily printed on a basic 3D printer and the motors are only 20$ each. Unlike others Poppy creatures, the electronic card is not integrated, which is very advantageous to manipulate.

This robot is the 'little brother' of the ergo robots used in the Ergo-robot experiment.

The list of parts to print is here and the 3D files there. You need 6 Dynamixel XL-320 (for example from here), a USB2AX to connect them to a computer and a small adaptation board.

After assembling your robot, try the discover quickstart, then have a look at the poppy_ergo_jr library.

Other interesting Poppy Creatures

Poppy right arm (work in progress)

poppy-6dof-right-arm is a Poppy creature based on a right arm of Poppy Humanoid, with 3 additionnal XL-320 motors at the end to improve the reach and agility of the arm.

The 3D-printed gripper can grab current objects, as shown in the video below:

This robot can be controlled with Python using Pypot as any Poppy creature, but also with **Matlab**. There is even a inverse kinematics algorithm allowing you to manipulate the arm in cartesian coordinate.

Find more info and the sources in https://github.com/poppy-project/poppy-6dof-right-arm.

The project was realized during an internship at INRIA Flowers by Joel Ortiz Sosa.

1.1.2 Version and changelogs

Version

TODO : update!

This is the version 1.1.0 of the Poppy documentation.

It contains the documentation for:

- Poppy Humanoid 1.0.1 hardware
- Poppy Torso 1.0.1 hardware
- Poppy Ergo Jr beta 6 hardware
- poppy_humanoid library version 1.1.1
- poppy_torso library version 1.1.5

- poppy_ergo_jr library version 1.4.0
- poppy.creatures library version 1.7.1
- pypot library version 2.10.0

Changelog

This is the first version of poppy-docs, but next version changes will be noted here.

1.1.3 Contribute!

The Poppy project's forum contains answer to your question, people that can help you and call for contribution. It is an important part of the project, so don't hesitate to ask, answer and contribute to it!

You can for example create a new Poppy creature, extend Pypot, create tutorials and practicals.

Call for contributions

Calls for contributions are regularly added in the forum.

Here is the list of current calls for contribution:

- Create new hand tooltips
- Develop a webapp for Poppy creatures
- User Interface to create choregraphies with Poppy, project lead by Thot
- Sensor board compatible with dynamixel protocol
- Backpack and batteries for Poppy Humanoid
- Extend the Poppy mini family
- Inflatable body enveloppe for Poppy Humanoid

Don't hesitate to help or to test the current projects!

Improve this documentation

TODO Exact precedure to be defined

Translation

You have no technical knowledge but you want to help by translating the doc? Many thanks! Go there TODO

Using Git and Github

All the sources of the Poppy project (software and hardware) are available on our Github.

If you want to modify one or our repositories, create a Github account (it's free of course) and fork the corresponding repo. When your modifications are done and tested, do a pull request with a meaningful name and comments if needed. The Poppy team will look at it as soon as possible and merge it if possible.

note

Please try to make pull request on unique coherant modifications or new features. If you work on several features at a time, please use branches and do separate pull requests to ease the life of libraries maintainers.

1.1.4 Build your own robot

Poppy Ergo Jr

Poppy Torso

1.1.5 Install the required softwares

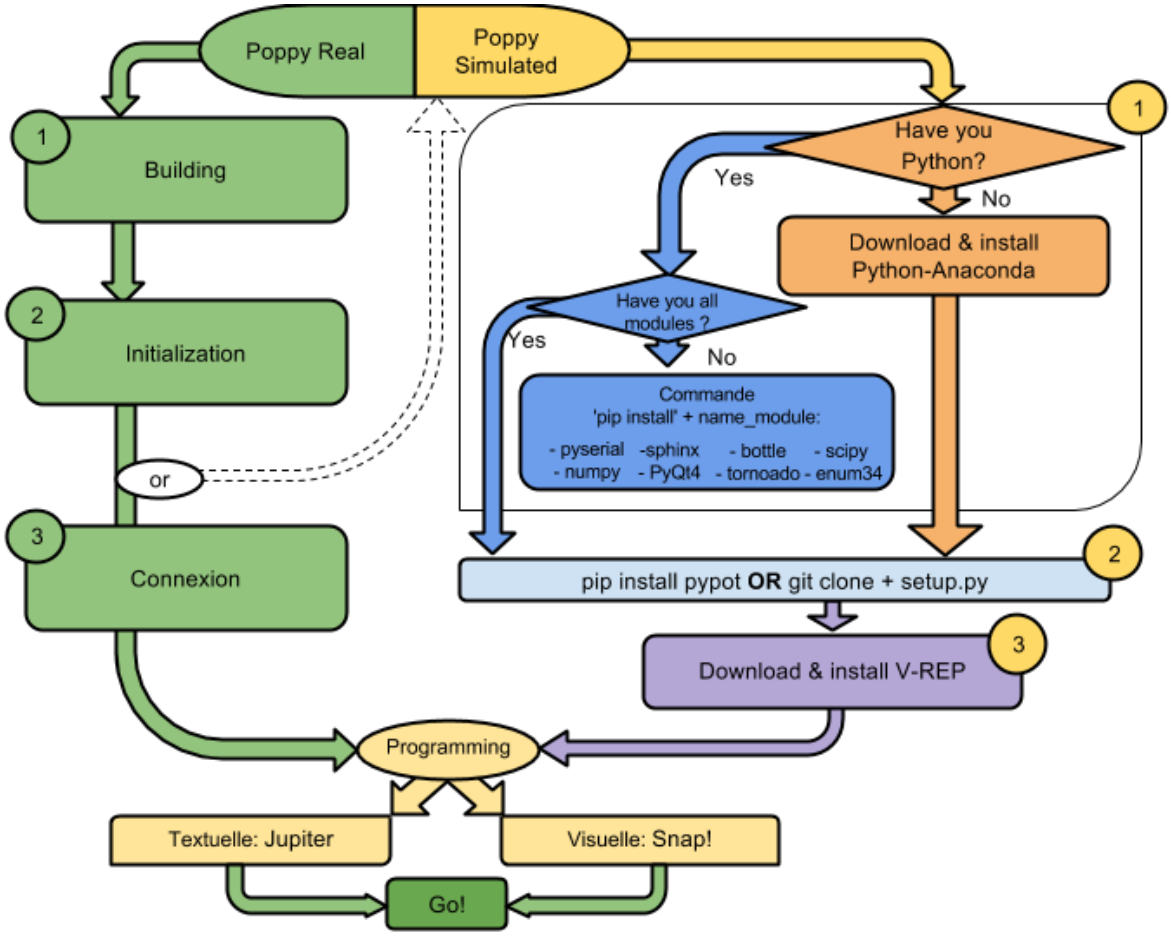
Once you have built your Poppy Creature or if you just want to try a simulated one, the next step is to start playing with it! Poppy Creatures can be controlled using the Python programming language and via Snap! a variant of the well-known visual programming language Scratch (or with other languages via the REST API, see this section for details).

Depending on what exactly you want to do - e.g. use Snap! to make your real robot moves or program in Python to interact with a simulated robot, different procedures must be followed.

Poppy Creature comes with an embedded board, all the needed softwares is already installed on their internal computer. They also comes with a webserver which allows you to directly program them - both in Python and Snap! - without needing to install anything. This method is described in the Control the robot from your web browser section. This is by far the easiest way but of course this is **only possible with a real robot and not in simulation**.

If you want to control a simulated Poppy Creature you will have to install the libraries on your own computer. The main Poppy libraries are written in Python and can thus be easily installed on any computer under Windows, Mac or Linux. The section Using a simulated robot describes those steps.

A diagram summarizing these different approaches can be seen below:



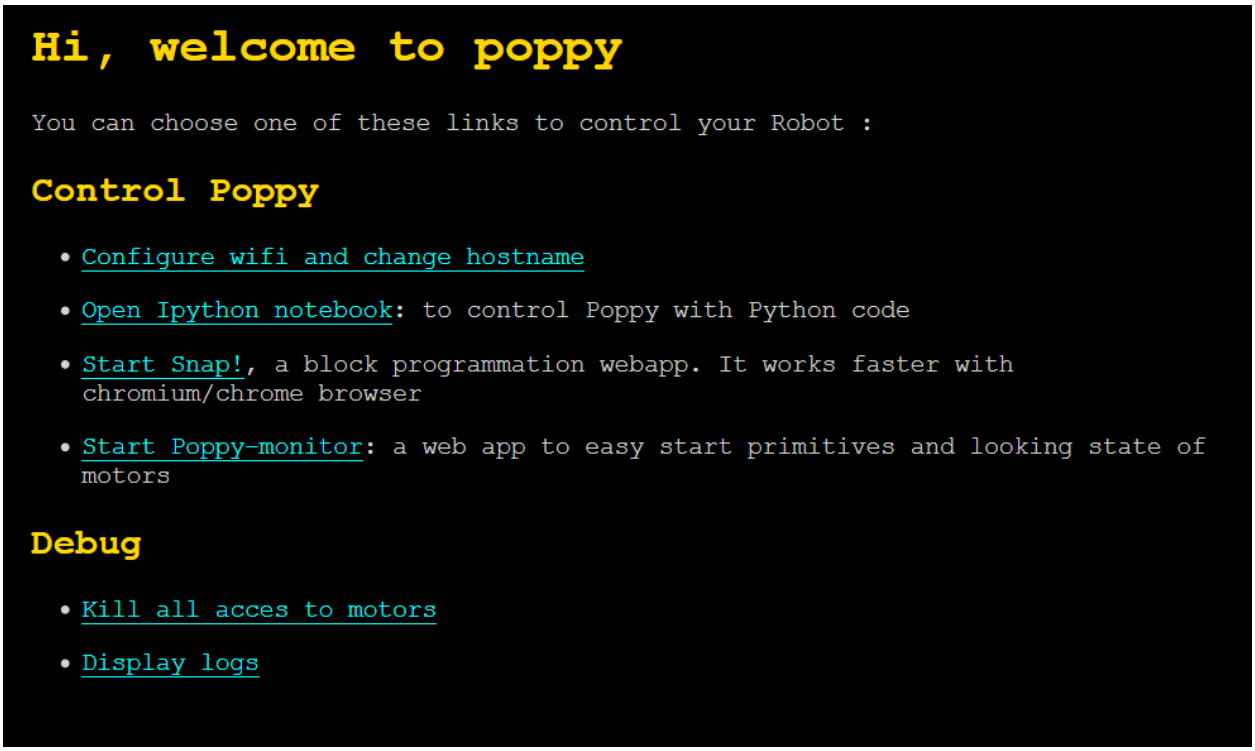
They are also other advanced approaches to directly control the robot from your own computer, but a solid knowledge of the linux and Python environments is required! Those approaches are described at the end of this page and detailed in the chapter Control your robot using your own computer.

Control a real robot from your web browser - no installation required

As explained above, Poppy Creature comes with an embedded board where Python and Poppy libraries are already installed and setup. They also comes with a server which permits a direct control of the robot from your web browser without needing to install anything on your computer.

To control your robot - either in Python or Snap! - from your computer a few steps are required:

- First, you need to make sure both your robot and your computer are on the same local network, e.g. on the same wifi or even better on the same ethernet network.
- Then, you need to either know the IP address of your robot or use the Bonjour services to automatically discover it - this service is already installed on most MacOS and Linux but must be installed on Windows.
- Finally, connect to http://poppy.local (replace *poppy.local* by the *IP address* if you do not use Bonjour) and you should see something similar to this:



From their you just have to follow the links and start programming your robot in Python or Snap! This is described in th next page.

Using a simulated robot

If you want to program a simulated version of a Poppy Creature you will have to install the Poppy software on your computer. Those libraries are written in Python and should be rather easily installed on any computer (Windows, MacOS and Linux).

Install Python and Poppy libraries

There are many ways to install Python and the needed packages, yet if you are not a Python guru we strongly recommend you to follow the process described below. You will avoid most of the common pitfalls. Of course if you are familiar with Python the classical *pip install poppy-ergo-jr* (or with any other creature name) you should be good to go.

- Install the Anaconda Python 2.7 distribution: it exists for Windows, Mac and Linux and comes with most of the dependencies pre-packaged. We recommend using Python 2.7 as this is our test environment, yet everything should work as well with Python 3.4 (or later).
- Install the Poppy libraries. You will need three main libraries:
  - pypot: it is the low-level library which handles all the communication with the motors and the sensors.
  - poppy-creature: it a light abstraction layer on top of the pypot library which defines everything which is common among all Poppy Creatures.
  - the library spefic for your creature such as poppy-ergo-jr, poppy-torso or poppy-humanoid. This is where everything specific to the creature is defined. You can install multiple of them if you want to use different creatures.

- To install them you have to open a command line. How to do this depends on your operating system. On recent Windows you can do it by following this guide for instance.
- Once the command line is opened simply type:

  ```
  pip install pypot
  ```

- Then:

  ```
  pip install poppy-creature
  ```

- And finally - you can obviously replace *poppy-ergo-jr* by the name of any other creature you want to install:

  ```
  pip install poppy-ergo-jr
  ```

To make sure everything is installed correctly you can type in the command line (replace PoppyErgoJr by the name of the creature you have installed: PoppyTorso or PoppyHumanoid):

```
python -c ``from poppy.creatures import PoppyErgoJr''
```

You can also install jupyter so you can use the notebooks:

```
pip install jupyter
```

If you do not see any error, it means everything is installed and you can continue and install the simulator. This is explained in the section Visualize your robot in a simulator.

Control it from your personal computer - the hard way

Warning: While the rest of this page was meant to be as accessible as possible, this section presents complex techniques reserved to advance users with a strong knowledge of linux and python environments.

The embedded board on Poppy Creature are simply tiny computer running Linux. They can be directly access via ssh or you can plug a keyboard and a screen and use it as any other computer.

You can also directly control the robot from your computer. You will need to install the same Python libraries as those needed if you use a simulated robot (pypot, poppy-creature and poppy-ergo-jr for instance).

The tricky parts concern the communication with motors. We use usb to serial communication, you will need either a USB2Dynamixel or a USB2AX. Depending on your operating system you may need a FTDI driver. On Linux make sure you have the necessary permissions to access the serial port, see here for details.

Warning: note that the ftdi driver on MacOS add a 16ms timeout which will make fast communication with motors impossible. If anyone know how to circumvent this please let us know! We recommend using Linux or Windows in the meantime. It also works great inside a VM.

Finally, accessing the sensors (e.g. the raspberry-pi camera) can be problematic from a computer but replacing it by any equivalent (any camera compatible with OpenCV) should do the trick.

For more details, you can refer to the exact procedure we follow for creating the image used in the embedded board to know all the details. It is described in the Setup the internal board section.

1.1.6 Program your robot using Python, Snap!, etc...

Connect your robot

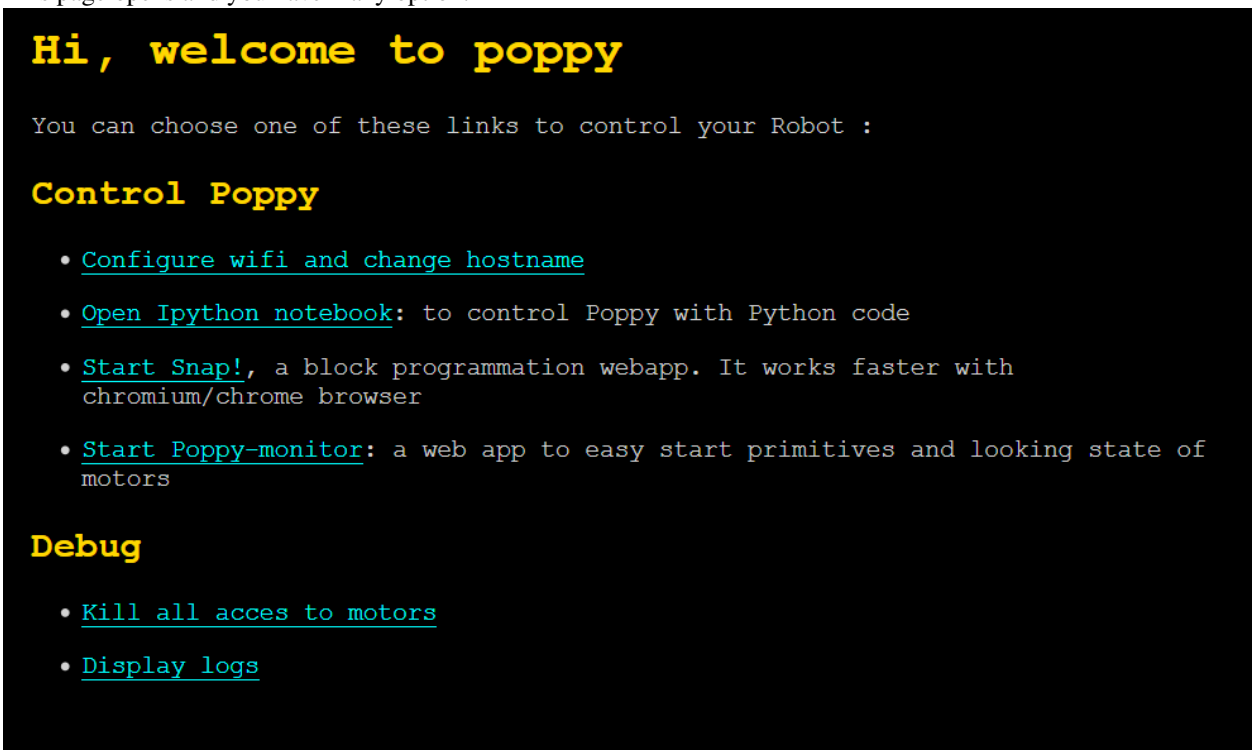
After install the required softwares (instruction here) you can connect your robot.

You need only a web-browser (like firefox or chrome) and be on the same network.

NB: in the first time a Poppy robot must connect via ethernet cable.

Enter this web url in your browser: poppy.local

This page opens and you have many option:



- Configure wifi: connecte wifi and enter the access key

- Change hostname: if you change a hostname, the url to acces in robot change also. *Exemple: if you change poppy to my_robot, the url to connect becomes my_robot.local*
- Open Ipython notebook (go here)
- Open snap (go here)
- Open Poppy monitor
- Debug option: display log and remove motor acces

Avec Python

The Poppy robots and in particular the pypot.robot.Robot objects from pypot make their best to allow you to easily discover and program you robot.

Where do I put my code?

If you are using a simulator, program directly in your computer, in a Python console (enter `python` in your terminal) or in any editor you like (Python IDLE...). Then launch it with the run button or use the command line `python myprogram.py`.

Otherwise, launch the IPython notebook server (via the webapp TODO or with this command in SSH `ipython notebook --ip 0.0.0.0 --no-mathjax --no-browser`) then connect with your browser to poppy.local:8888 TODO ipython screenshot Create a new notebook and put your code into it.

Last solution, SSH into your robot (using Putty or `ssh poppy@poppy.local`) and open an editor (`vim` by default on the Odroid) to put your code directly in the robot. Note that if you want to use a more advanced editor, you can use the -X option while connecting in SSH to get XWindows back on your desktop computer.

Create the robot object

The first (and more difficult!) step is to create this robot object.

If you have a physical Poppy Humanoid, Poppy Torso or Poppy Ergo Jr, you can directly use the corresponding libraries and create your robot with (let's call the robot *poppy*):

```
from poppy_humanoid import PoppyHumanoid
poppy = PoppyHumanoid()

from poppy_torso import PoppyTorso
poppy = PoppyTorso()

from poppy_ergo_jr import PoppyErgoJr
poppy = PoppyErgoJr()
```

Otherwise, if you use a custom robot defined in a configuration file, use (see here):

```
import pypot.robot
poppy = pypot.robot.from_json(`my_config.json')
```

Motors

Now that we are connected to the robot, we can list all available motors:

```
print(poppy.motors)
```

You get something like:

```
[<DxlMotor name=l_elbow_y id=44 pos=-0.0>,
 <DxlMotor name=r_elbow_y id=54 pos=-0.0>,
 <DxlMotor name=r_knee_y id=24 pos=0.2>,
 <DxlMotor name=head_y id=37 pos=-22.7>,
 <DxlMotor name=head_z id=36 pos=0.0>,
 <DxlMotor name=r_arm_z id=53 pos=-0.0>,
 <DxlMotor name=r_ankle_y id=25 pos=1.6>,
 <DxlMotor name=r_shoulder_x id=52 pos=1.1>,
 <DxlMotor name=r_shoulder_y id=51 pos=0.0>,
 <DxlMotor name=r_hip_z id=22 pos=0.1>,
 <DxlMotor name=r_hip_x id=21 pos=1.2>,
 <DxlMotor name=r_hip_y id=23 pos=-0.0>,
 <DxlMotor name=l_arm_z id=43 pos=-0.0>,
 <DxlMotor name=l_hip_x id=11 pos=-0.0>,
 <DxlMotor name=l_hip_y id=13 pos=-0.1>,
 <DxlMotor name=l_hip_z id=12 pos=0.1>,
 <DxlMotor name=abs_x id=32 pos=0.0>,
 <DxlMotor name=abs_y id=31 pos=0.4>,
 <DxlMotor name=abs_z id=33 pos=0.0>,
 <DxlMotor name=l_ankle_y id=15 pos=0.8>,
 <DxlMotor name=bust_y id=34 pos=0.8>,
 <DxlMotor name=bust_x id=35 pos=0.2>,
 <DxlMotor name=l_knee_y id=14 pos=0.0>,
 <DxlMotor name=l_shoulder_x id=42 pos=0.4>,
 <DxlMotor name=l_shoulder_y id=41 pos=0.0>]
```

We can make it more readable by extracting the motor's name and position:

```
for m in poppy.motors:
  print(`motor {} in position {}'.format(m.name, m.present_position))
```

The motors are grouped into motor groups:

```
for group in poppy.alias:
    print (``motor group ``, group)

# assuming ``head'' is one of the motor groups (use ``top'' for a ergo_jr)
for m in poppy.head:
    print(`motor {} in position {}'.format(m.name, m.present_position))
```

Motors are said compliant if they do not apply torque to reach their goal position. They are safe and can be moved by hand. To have the robot move by itself, you must remove the compliance.

> **warning**
>
> Before removing compliance, be sure each motor is in its allowed angle range, otherwise, the robot may move very quickly to get back to an *allowed* position

You can do it for the whole robot, for a motor group or for each motor:

```
# set the whole robot compliant
poppy.compliant = True

# set the whole motor group non-compliant
poppy.head.compliant = False

# set thone motor non-compliant
```

```
poppy.l_shoulder_x.compliant = False
```

Now that compliance is removed, let make the robot move! Put it in an open place, far from your coffee, fingers and screen and let's go. The `present_position` of a Dynamixel motor is its angular position (in degree) sensed by the integrated rotationnal encoder. Its`goal_position` is the angular position we ask it to achieve and the motor will therefore move toward it:

```
import time

poppy.head_z.goal_position = 20.
poppy.head_y.goal_position = 10.

time.sleep(2) # wait 2 seconds

poppy.head_y.goal_position = -10.
```

See the motors documentation for more advanced control.

Sensors

As Poppy Creatures are made of both motors and sensors, you can naturally also access those sensors.

First, you can retrieve all sensors attached to your robot via:

```
print(poppy.sensors)
>>> [<pypot.sensor.imagefeature.marker.MarkerDetector object at 0x7fb11f05f4d0>,
     <pypot.sensor.camera.opencvcam.OpenCVCamera object at 0x7fb11f042650>]
```

You can also get their names:

```
print([sensor.name for sensor in poppy.sensors])
>>> [`marker_detector', `camera']
```

and then directly use this name to access the sensor:

```
print (poppy.camera)
>>> <pypot.sensor.camera.opencvcam.OpenCVCamera object at 0x7fb11f042650>
```

The present sensors vary depending on which Poppy Creature you are using. For instance, the ErgoJr has:

- a camera
- image feature detection (marker detection in the example above)

You can then introspect each sensor to get available registers:

```
print(poppy.camera.registers)
>>> [`frame', `resolution', `fps', `index']
```

And then read the value you are interested in:

```
print(poppy.camera.resolution)
>>> [640, 480]
```

More details on the sensors and how you can add extra sensors are given in the extending sensors section.

> **note:** We separate motors and sensors even if this is kind of an artificial definition here. Indeed, we used servomotors which can be seen both as motors and sensors.

Go further

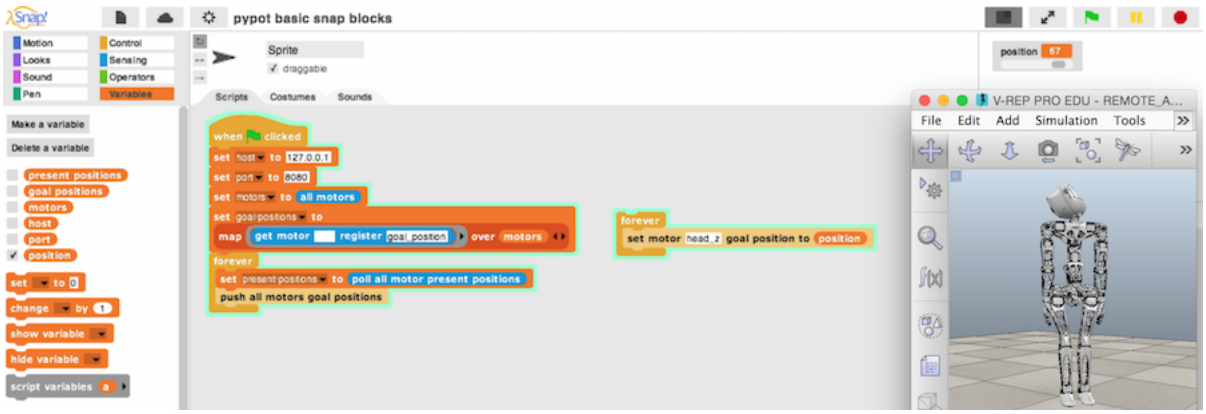
Next step is to use primitives, which is decribed in this tuto.

Control a Poppy robot using Snap!

Introducing Snap!

snap visual programming language is a "very powerful visual, drag-and-drop programming language. It is an extended reimplementation of Scratch (a project of the Lifelong Kindergarten Group at the MIT Media Lab) that allows you to Build Your Own Blocks". It is an extremely efficient tool to learn how to program for kids or even college students and also a powerful prototyping method for artists.

Snap! is open-source and it is entirely written in javascript, you only need a browser connected to the Poppy Creature webserver. No installation is required on your computer!



An introduction to this laguage can be found in the Snap! reference manual.

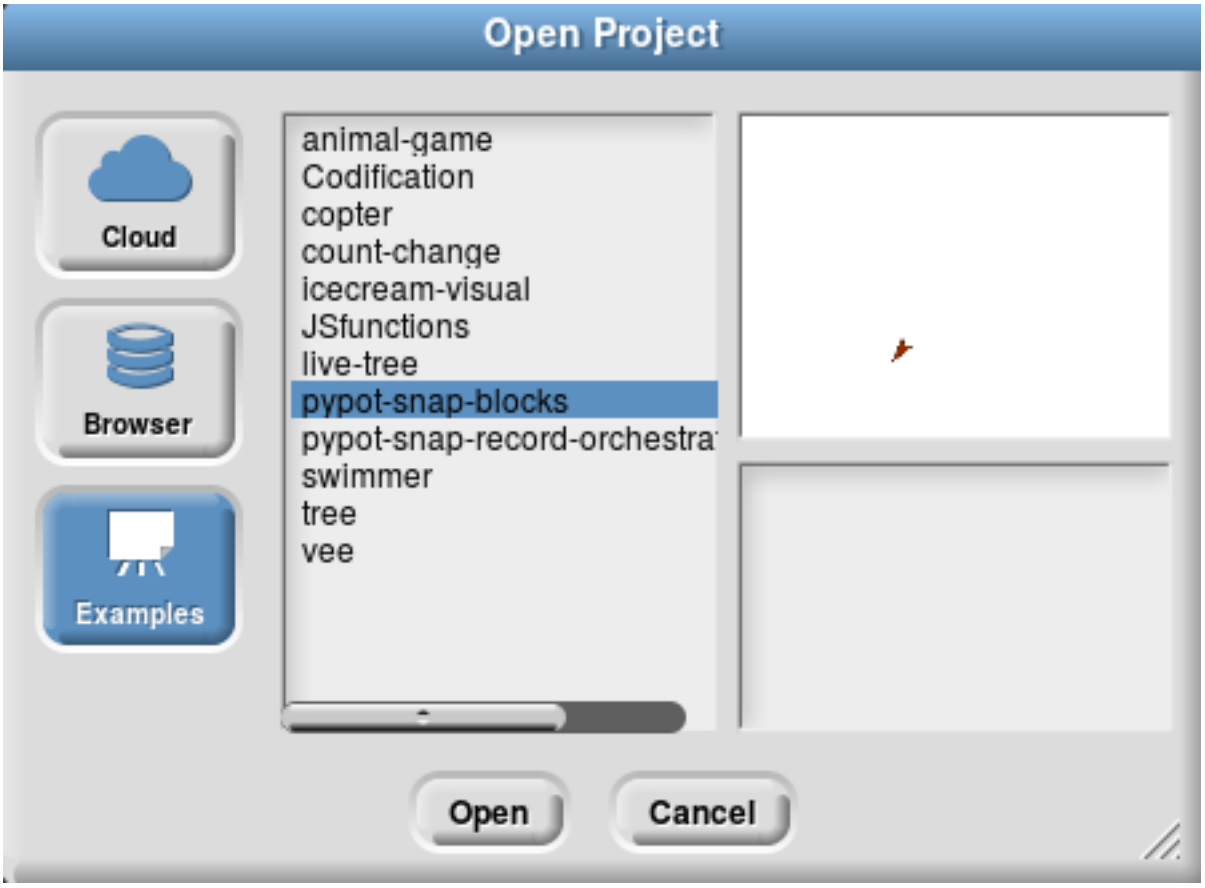
Create the Snap! server

There are several ways of creating a Snap! server to control your robot.

Warning! At some point of this tutorial, you will have to import Poppy Humanoid's specific Snap! blocks. This may take a few minutes, so keep the script running if your browser asks what to do.

Start the server on the robot The easiest way of controlling your Poppy robot with Snap! is to use the webapp TODO. This will launch the server and use the Snap! software installed on the robot.

Then hit the folder button, select open...->example and choose pypot-snap-block.



Alternately (if you don't have the webapp), you can use the following command inside the robot:

```
poppy-snap poppy-humanoid --no-browser
```

The command gives you an URL (something like http://snap.berkeley.edu/snapsource/snap.html#open:http://-ROBOT-IP-:6969/snap-blocks.xml). Open this URL in your web browser.

Start the server on your computer Use this method if you use the motors directly linked to your computer:

```
from poppy_humanoid import PoppyHumanoid
poppy = PoppyHumanoid(use_snap=True)
poppy.snap.run()
```

You can even use simultaneously Snap! and V-rep:

```
from poppy_humanoid import PoppyHumanoid
poppy = PoppyHumanoid(simulator='vrep', use_snap=True)
poppy.snap.run()
```

Leave the Python script running. In you web browser, open the following URL:

http://snap.berkeley.edu/snapsource/snap.html#open:http://127.0.0.1:6969/snap-blocks.xml

Last solution: you can even, if you don't have a reliable internet connection when you use Snap!, download and install Snap! directly on your computer from here.

Then, open the snap.html file with your web browser and use the folder button->import.. to import the Poppy specific blocks located in *pypot/pypot/server/snap_projects*.

Controlling the robot

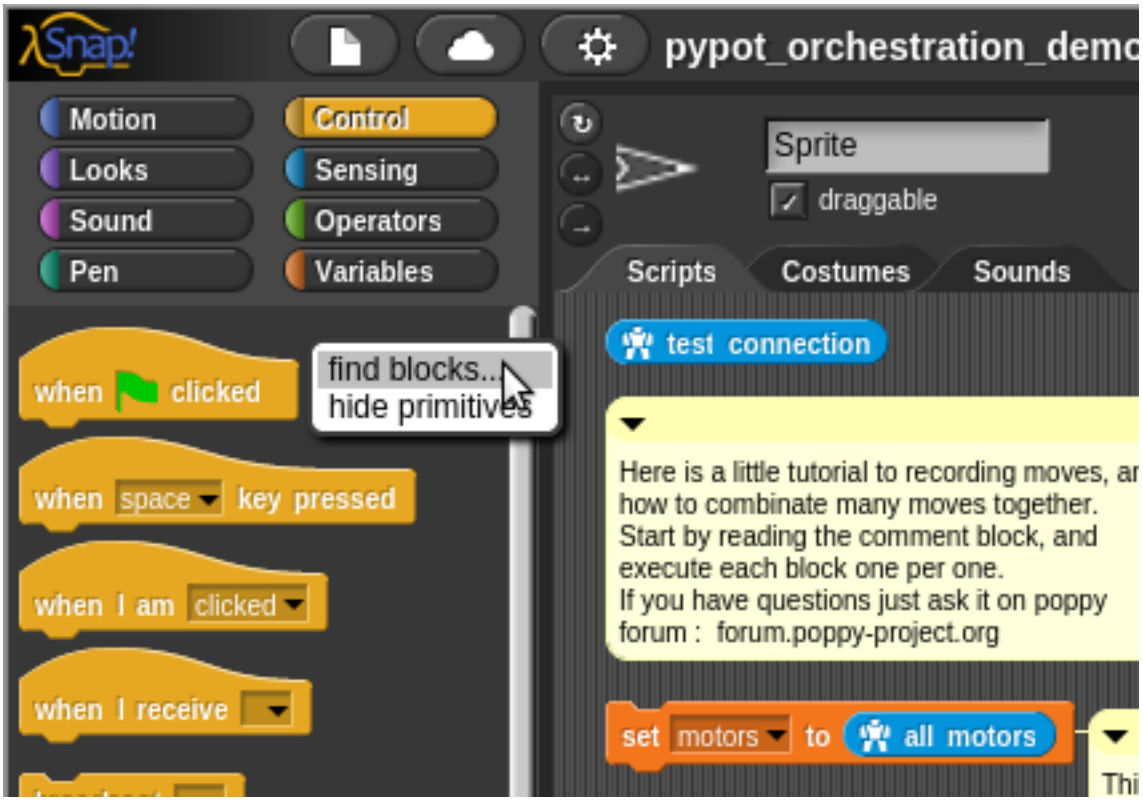
Pypot Snap! blocks The Poppy Snap! blocks are the following:


test connection
all motors
motors from alias
get motor motor_name register
get motor motor_name present position
motor index
all aliases
all primitives
all records
motor(s) goto position 0 in 0 seconds
set motor(s) compliant
set motor(s) register to
create move move name with motor(s)
stop record move name
start record move name
stop play move name
play move name | speed x 1
play in reverse move name | speed x 1
play concurently moves
play sequentialy moves

Those blocks can be used to respectively:

- test if connection with poppy robot is working well
- get a list of all motors name
- get a list of all motors refered by an alias
- get the value of a register motor (e.g. get motor “head_z” register “present_load”)
- get the index of a motor
- get all alias avaible for the current robot
- set a motor position in a specified time
- turn a motor compliant or not
- set a register of a motor (e.g. set motor “head_z” register “present_load” to 10)
- create/attach a move to some motors (you have to create a move before to record or replay it)
- stop the record of a move
- start the record of a move
- play a move at a defined speed
- play a move in reverse at a defined speed
- play concurently many moves
- play sequentialy many moves

You can easily see all blocks relative to poppy in Snap! with the “find blocks” feature. You have to right-click in the left part of Snap! page and select “find blocks”:



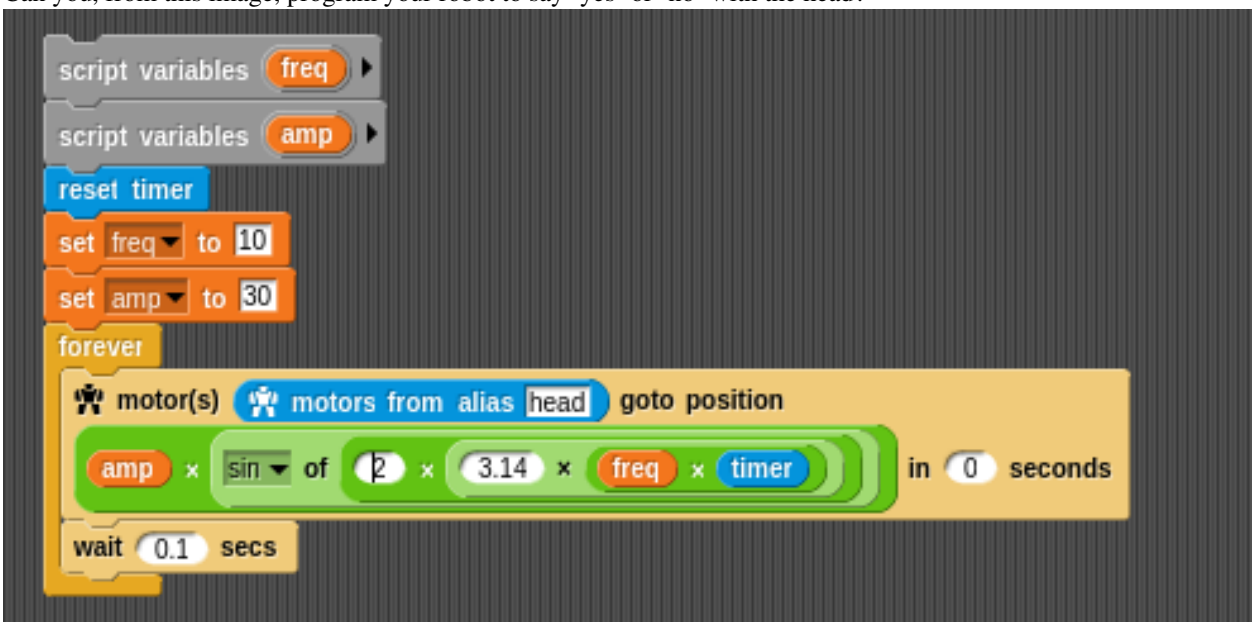
Use a slider to move a motor To control a motor via a slider you need to make a variable - we will call it head position.

Then right click on it and use the slider option. Change the slider min/max to (-50, 50).

Then, connect it to a motor: use the motor(s) goto position block and put it inside a forever loop. Add a wait for performance issue.

Example: playing a sinus on a motor Having a motor position follow a sinus function is very useful to get smooth periodic moves, as waving with the hand or saying 'no' with the head.

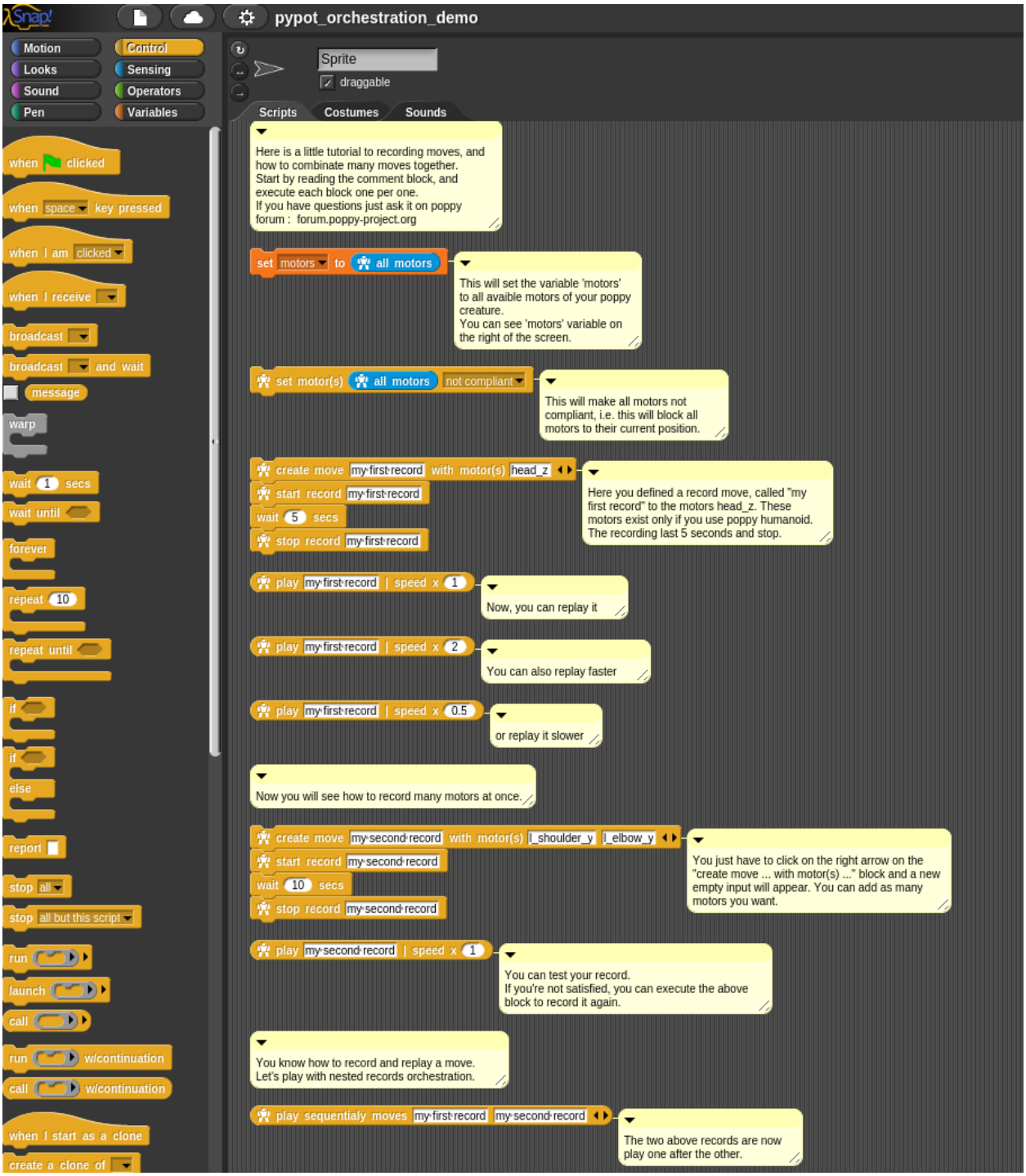
Can you, from this image, program your robot to say 'yes' or 'no' with the head?



Record and play moves using Snap!

If you opened the Snap! server using the webapp, you can directly load the example called pypot-snap-record-orchestration-demo instead of poppy-snap-blocks to find a ready-to-use Snap! project dedicated to the record and replay of moves.

Otherwise, use the folder button->import.. and select *pypot/pypot/server/snap_projects/pypot-snap-record-orchestration-demo.xml*



Accessing your robot through the Rest API

As Poppy Creatures are made to be easily integrated into educational, research or artistic projects, it is central that they can be easily connected with the external world. Thus, we desgined a REST API which permits the access of all main features of a Poppy Creature through HTTP requests.

Thanks to this REST API it is easy to:

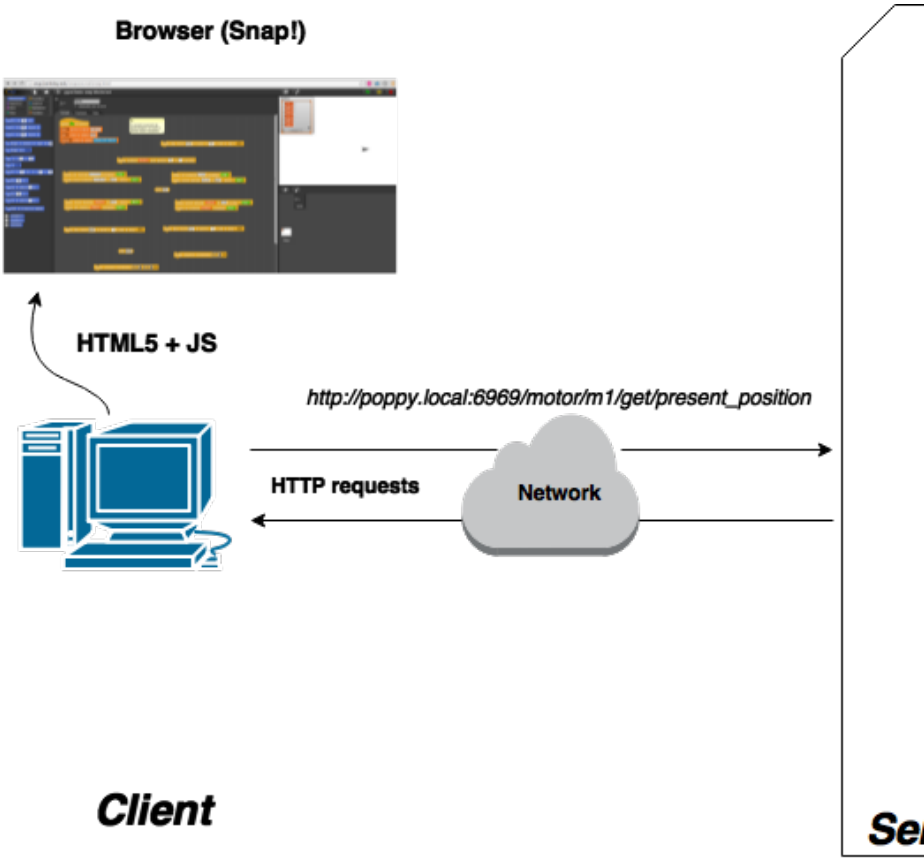
- control your robot through any connected device (such as a smartphone),
- build bridges to control a Poppy Creature with any language (bridges already exist for Matlab and Processing thanks to our awesome contributors!),
- connect multiple Poppy Creatures so they can easily interact.

  Warning: The REST API is still under active development and should be improved and stabilized soon.

The REST API gives you access to most of the Poppy Creature's sensors and motors registers. You can retrieve and send values to the moros. You can also control primitives (start and stop them for instance). The exhaustive list can be seen here. A few examples are given in the table below:

| Functionality | URL | |———————————————————|———————————————————| | Get motors list. | GET /motor/list.json | | Get the 'present position' of the motor 'm3'. | GET /motor/m3/register/present_position | | Start the 'dance' primitive. | GET /primitive/dance/start.json |

Note: All answers are returned using JSON format.



This image sumarizes how the REST API works:

1.1.7 Visualize your robot in a simulator

Introducing V-rep

V-rep is a physical simulator that allows you to simulate robots in an enviroment with gravity, contacts and friction.

It is compatible with Windows, Mac and linux and can be installed from here.

It is free to use if you are from the education world and you can get a limited version otherwise.

Warning: a physical simulation always asks for quite a lot of computation, so try to run it on a not-too-old computer.

Start a simulation with Poppy Humanoid

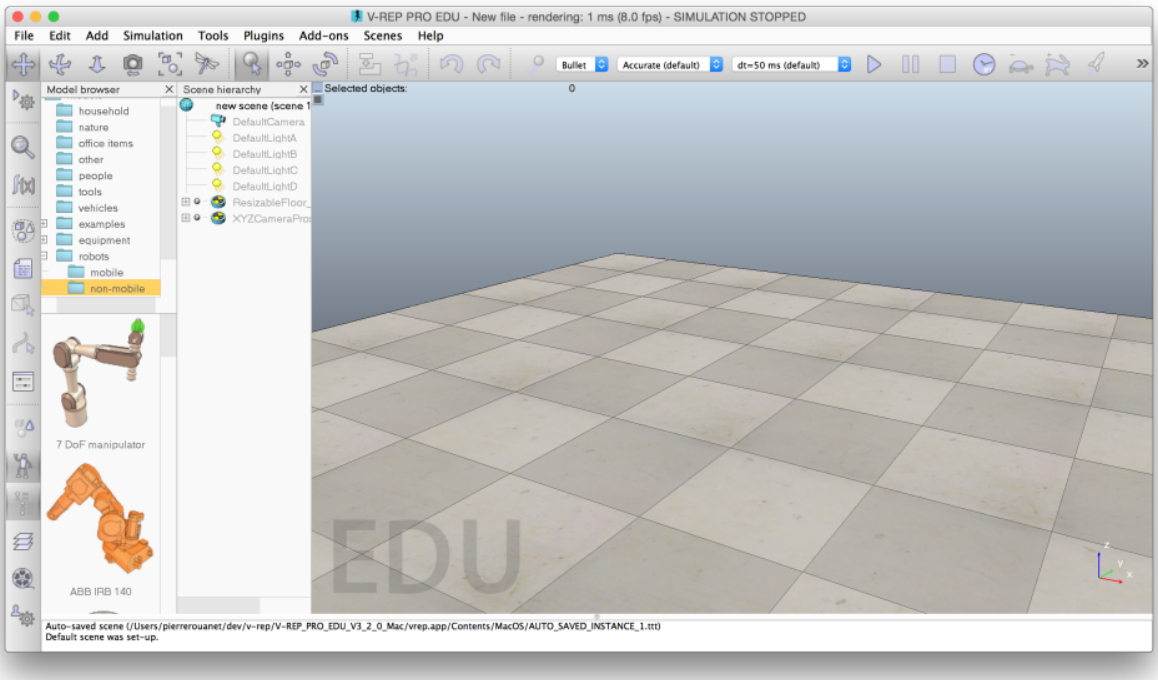
Once you have V-rep and poppy-humanoid installed on your computer (see here for installing poppy-humanoid and therefore pypot on your desktop computer), you need to follow these two steps:

Start V-rep

On Windows, find the .exe file and execute it. On linux and mac, open a terminal, go to the folder where v-rep is install and launch it with:

```
./vrep.sh
```

You get a window that looks like this:



You should get an empty world with a floor and a tree structure of the elements of the world on the right.

You can explore the world by drag-and-dropping the simulated world and you can start/pause/stop the simulation with the up-right corresponding buttons. As v-rep uses a lots a computing power, it is advised to pause the simulation while your robot don't move.

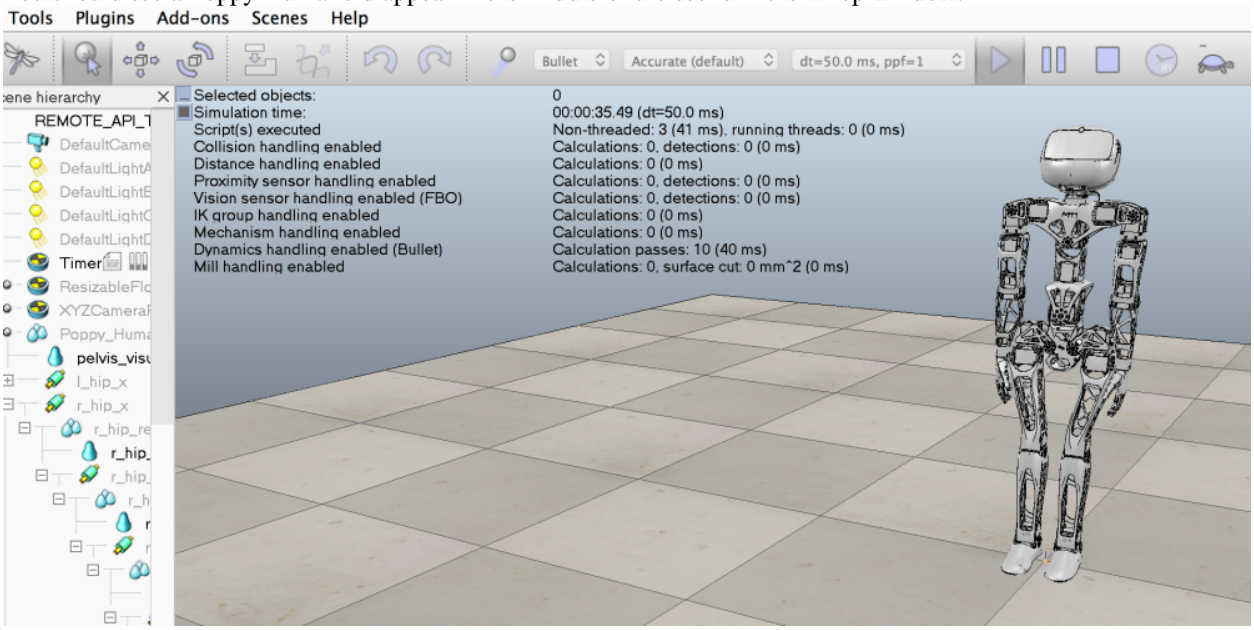
Launch a scene containing a Poppy Humanoid robot

The 3D model for the Poppy Humanoid robot is included in the poppy-humanoid package.

So, all you have to do is open a Python console or start a Python file and start it with:

```
from poppy_humanoid import PoppyHumanoid
poppy = PoppyHumanoid(simulator='vrep')
```

You should see a Poppy Humanoid appear in the middle of the scene in the V-rep window:



The *poppy* object that we just created can now be used exactly as a PoppyHumanoid object created for a physical robot.

For example, you can test it with:

```
# print all motors
print poppy.motors
#ask a new position for the head
poppy.head_z.goal_position = -10

#wait a bit
import time
time.sleep(2.)

#print head_z position
print poppy.head_z.present_position
```

If the simulated Poppy Humanoid turns its head, well done, it works! Now you can follow this tutorial to learn how to control it!

You may want at some point to go back to the initial state of the simulation. Here is the command:

```
poppy.reset_simulation()
```

Troubleshooting and advanced usage

See here for a view on lower-level link between your robot and V-rep.

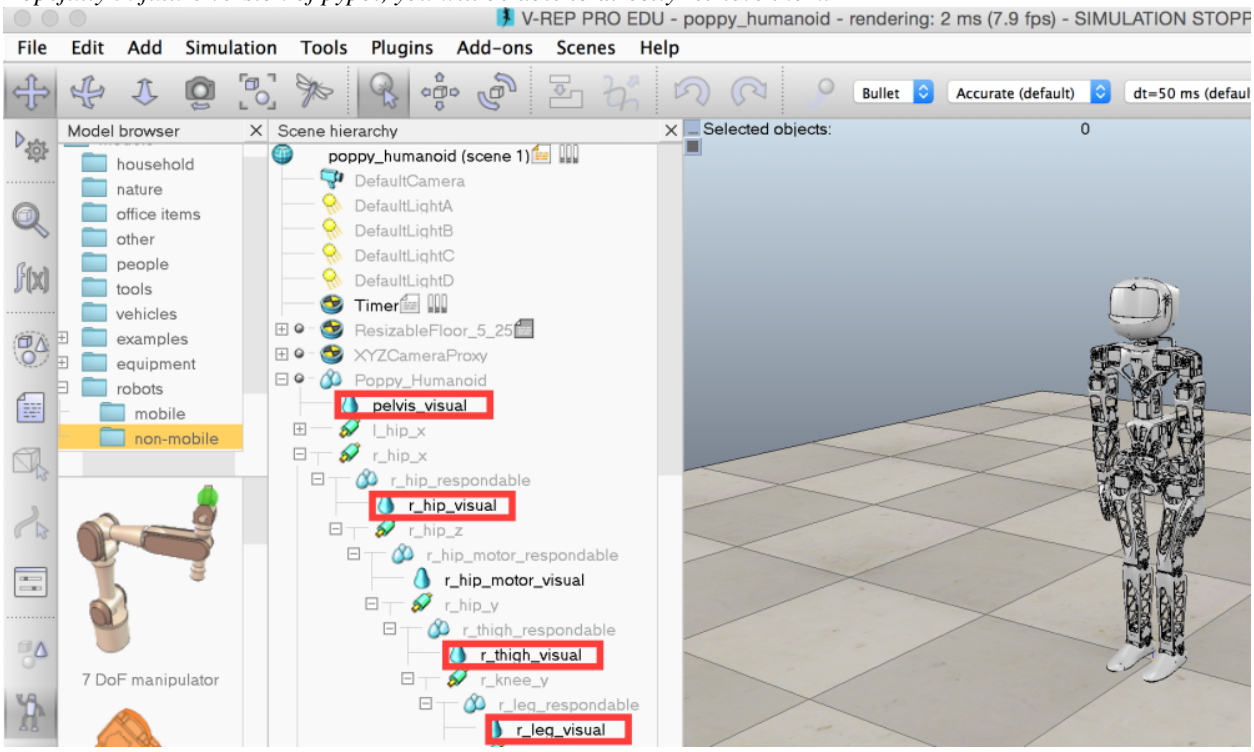
TODO

- what if it doesn't connect ?
- what if it freezes ?
- how to create a scene with objects ?
- how to add another creature ?

Usage example: determine reachable space for the arm

Using a V-REP simulated robot, you can easily retrieve an object position and orientation. You just need to know its name in the vrep scene.

Note: at the moment to know the name of object in the vrep scene, you have to look for them in the v-rep window. Hopefully in future version of pypot, you will be able to directly retrieve them.



For instance, to get the 3D position of the left hand, you just have to do:

```
poppy.get_object_position(`l_forearm_visual')
```

You get a list of 3 positions in the V-REP scene referential (the zero is somewhere between Poppy Humanoid's feet). You can use any object as referential and thus get the left forearm position related to the head for instance:

```
poppy.get_object_position(`l_forearm_visual', `head_visual')
```

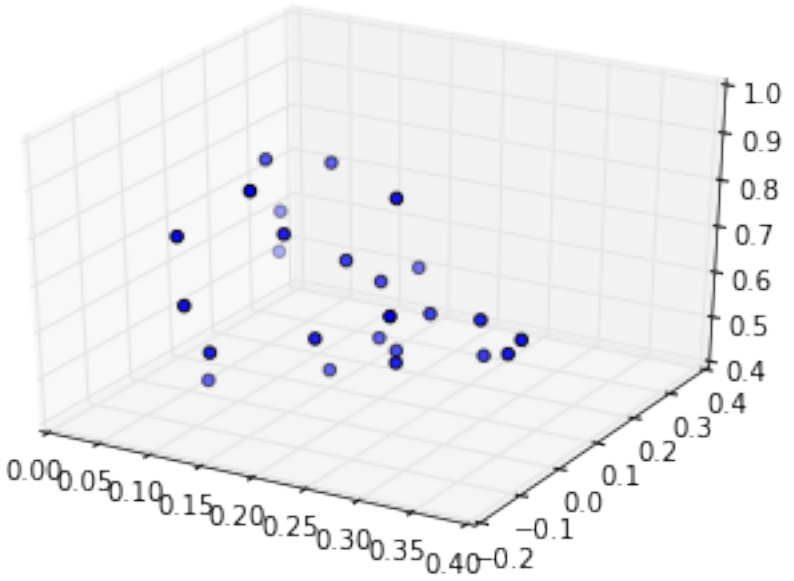
To discover the reachable space of the left hand of the robot (with respect to its head), you can for example generate many random positions for the arm (here, 25) and store the reached positions:

```
import random

reached_positions = [] #we will store the positions here

for _ in range(25):
    poppy.reset_simulation()

    # Generate a position by setting random position (within the angle limit) to each joint
    pos = {m.name: random.randint(min(m.angle_limit), max(m.angle_limit)) for m in poppy.l_
    print ``Getting forearm position when motors ar at '',pos

    #make the robot reach position
    poppy.goto_position(pos, 2., wait=True)
```

```
    #get and store forearm position
    reached_pt.append(poppy.get_object_position(`l_forearm_visual'))
```

Now matplotlib or any other plot library can help you visualize the data:

```
from mpl_toolkits.mplot3d import Axes3D

ax = axes(projection=`3d')
ax.scatter(*array(reached_pt).T)
```



TODO: can someone confirm that code and imports are OK ?

CHAPTER

TWO

ADVANCED SECTIONS

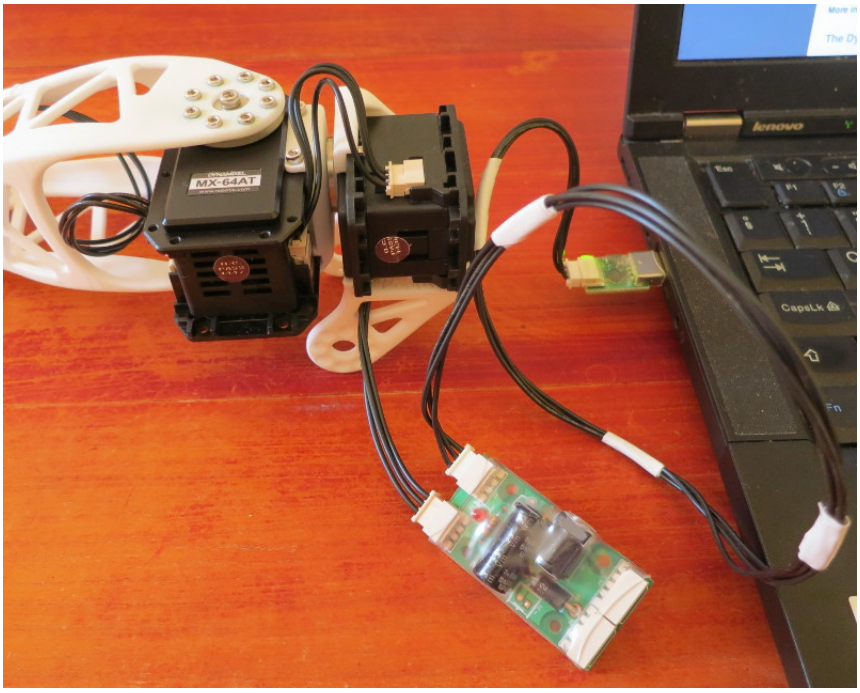
2.1 Assembly Guides

2.1.1 Working with Dynamixel servomotor

Dynamixel hardware

The Poppy Humanoid robot is mainly built with MX-28AT Dynamixel servomotors (MX-28T are the previous version and can be used without any problem). The other servomotors are MX-64T (bigger and stronger) and AX-12A (smaller, used for the head).

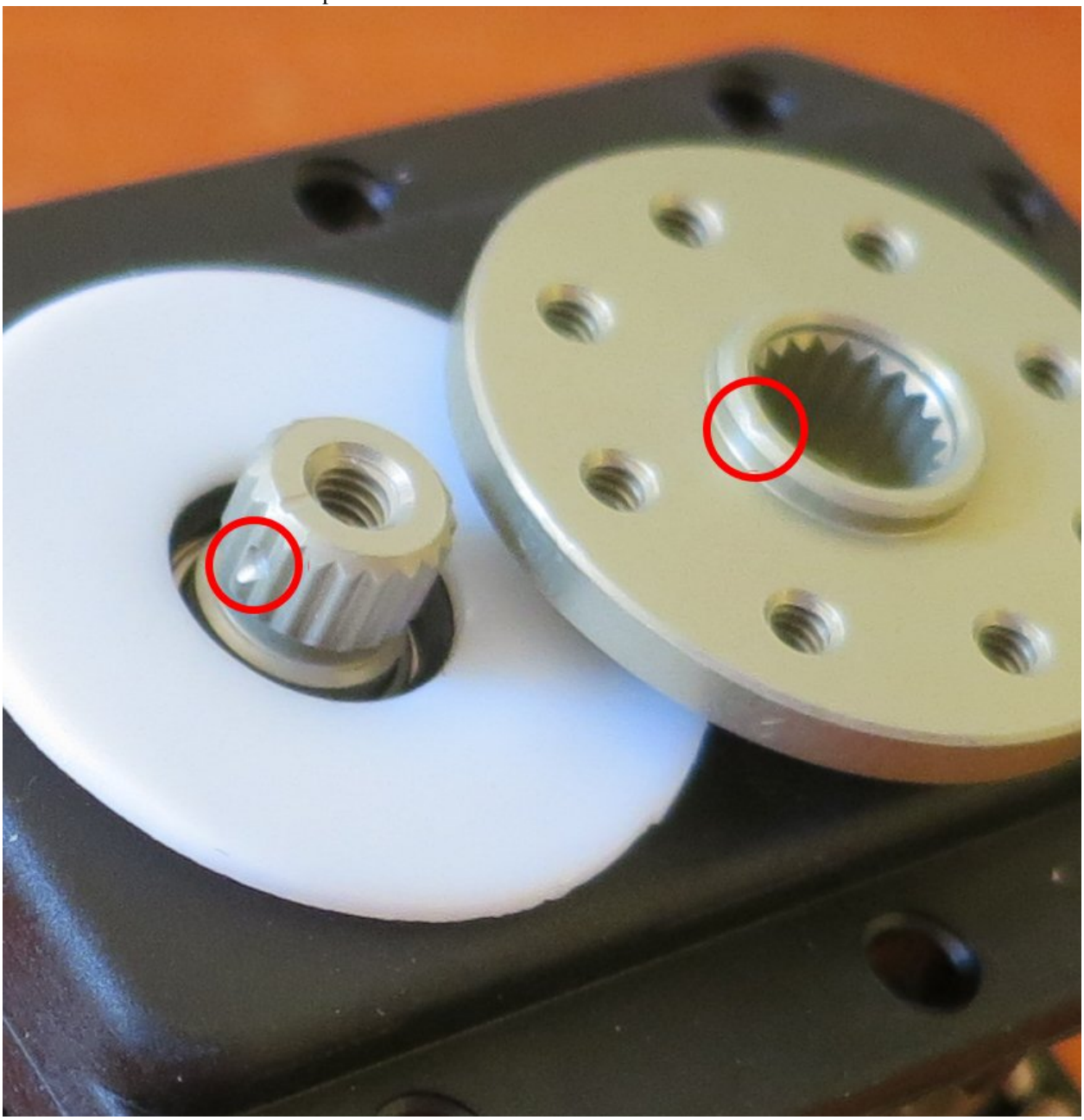
Each Dynamixel servomotor embeds an electronic board allowing it to receive different kind of orders (about goal, torque...) and communicate with other Dynamixel servos. Therefore, you can chain up several Dynamixel servomotors (each with a different ID) and command them all from one end of the chain: each servomotor will pass the orders to the next one.



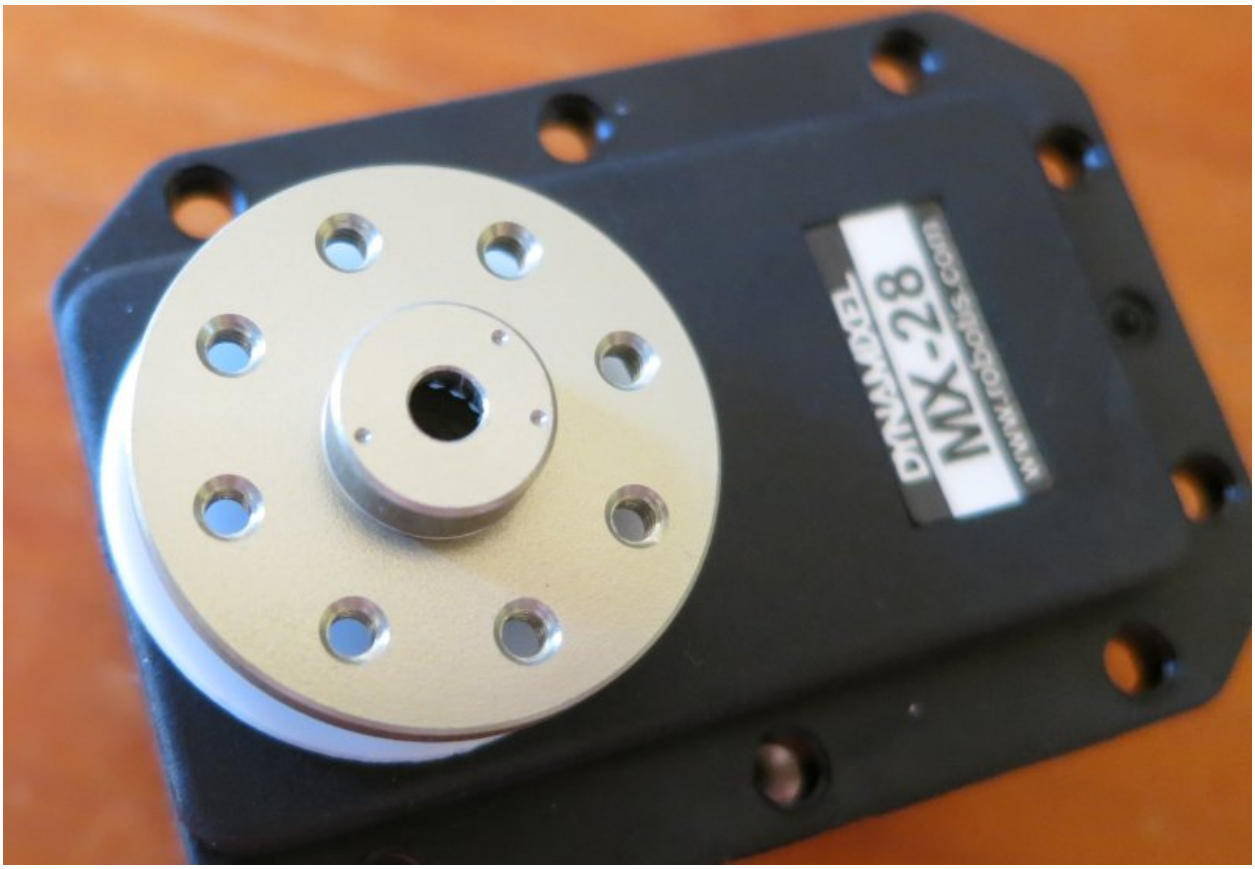
Putting the Dynamixel horns to zero

When you receive your Dynamixel servomotors, the horns are not mounted. They are included in the packaging if the servo is packaged alone or packaged separately for 6-pieces bulks (see next section to know what horn goes to what servo).

When putting the controlled horn, be very careful to **put the dot on the horn at the same point than the dot on the servo axis**. Once the horn is put, it is most of the time **impossible to remove** ! This will ensure that the zero position of the servo matches with the zero position of the structure around.



On the outside of the horn, you also have three dots indicating the orientation. You should find the same three dots on structural parts, so be sure to match them.



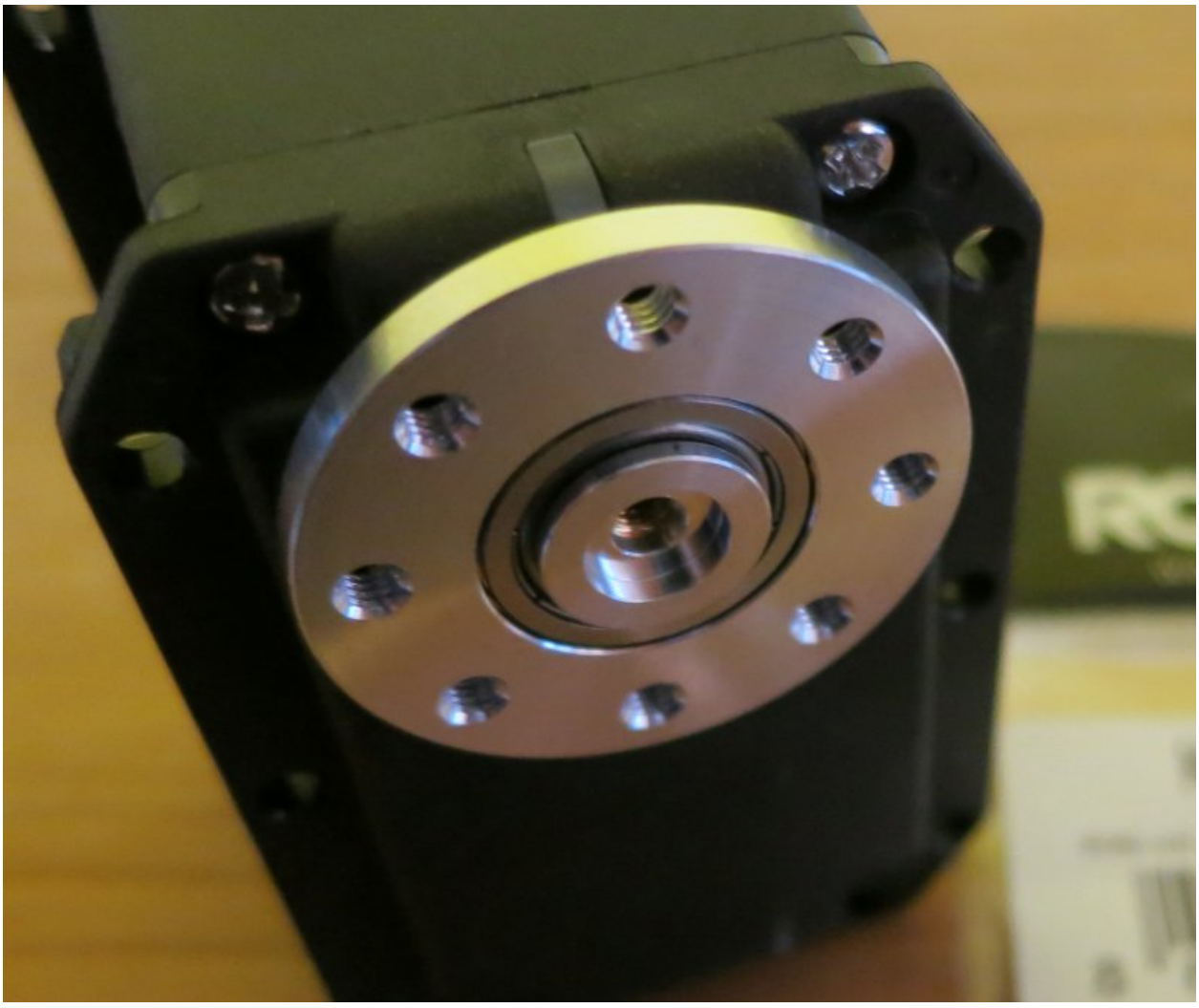
Horns of MX-28 and MX-64

On each Dynamixel servomotor apart from the AX-12A, you will have to mount a horn to the motor axis. Most of the time, you will also have to mount a free horn on the opposite side to provide better fixation points for the structure parts.

To mount the main horn, put the plastic ring (white or black) and drive the horn on the axis. **Be careful of the zero when putting the main horn!** Then put thread locker on the big screw and screw it in the middle.

Main horn mounted on a MX-28

For the free horn, first clip the ball bearing and the cap on the side without shaft shoulder. Then put the horn on servomotor (with shaft shoulder on servo side). Put thread locker on the big screw and screw it. The horn should turn freely.



Free horn mounted on a MX-64

Quick reminder of horn names and screw sizes:

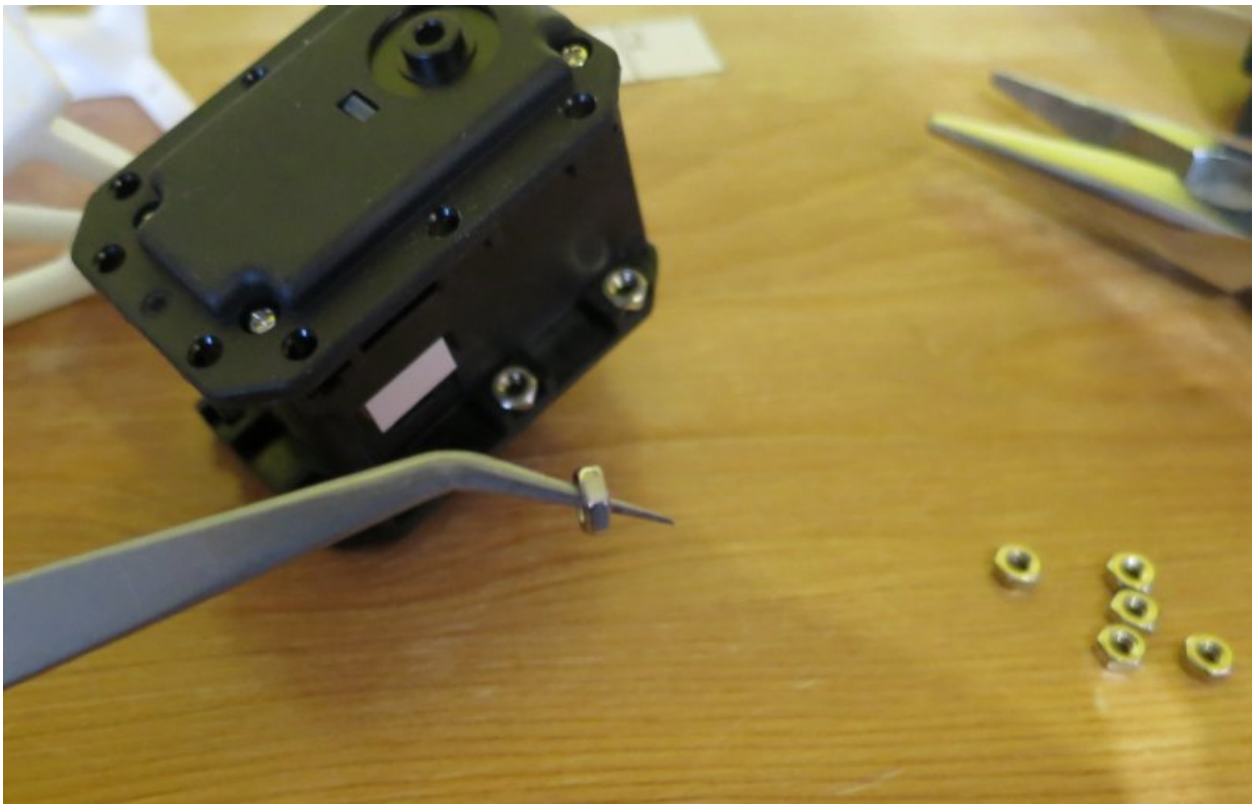
| Servomotor | main horn | free horn | big horn screw | horn screws | case screws| |—————| :————: | :————: | :————-:| :————-: | :————-:| |AX12-A | none | none | M3x10mm | M2 | M2 | MX28 | HN07-N101 | HN07-I101 | M2.5x8mm | M2x3mm | M2.5x6mm | MX64 | HN05-N102 | HN05-I101 | M3x8mm | M2.5x4mm | M2.5x6mm |

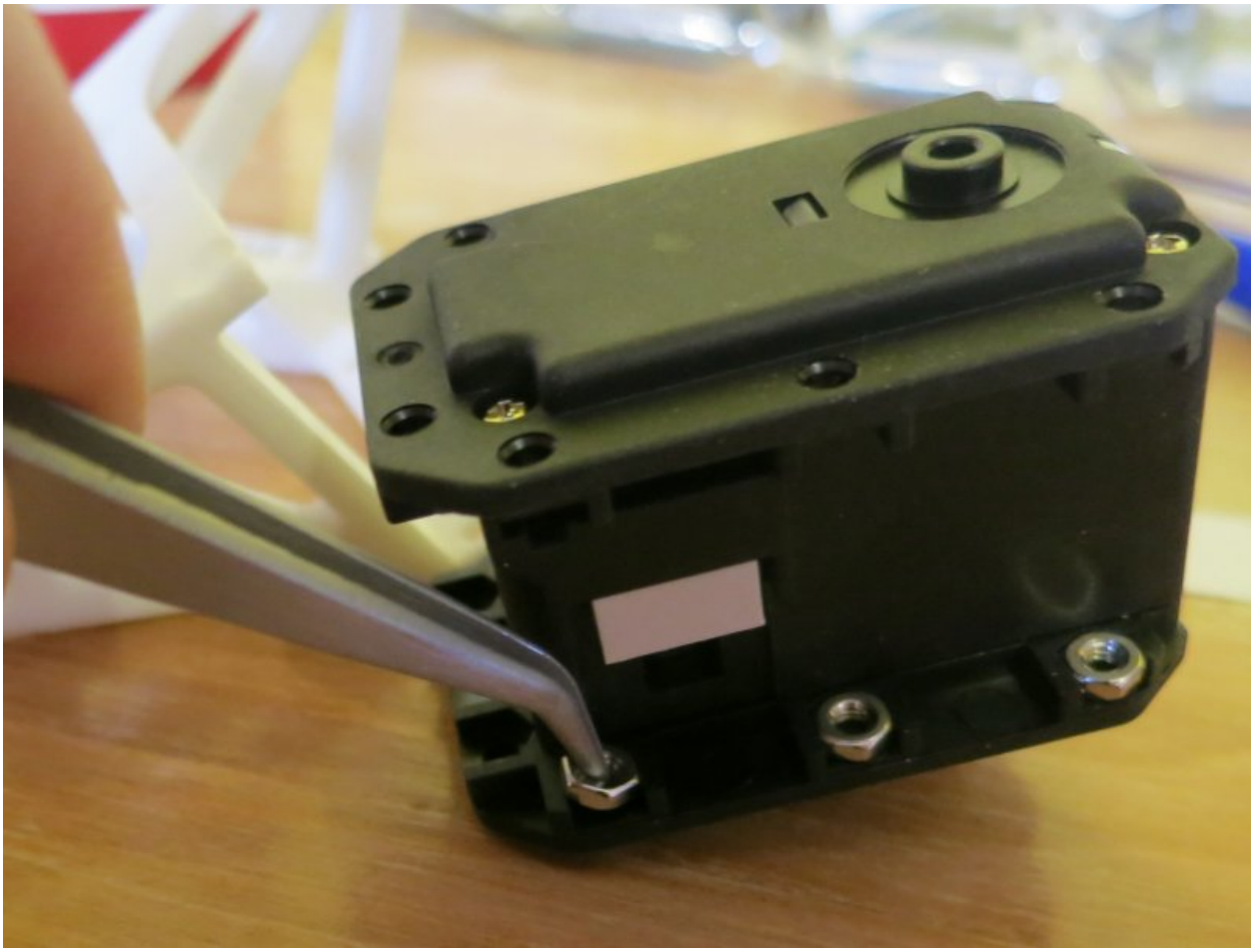
You need an allen wrench of size 1.5mm for M2 screws, 2mm for M2.5 screws and 2.5mm for M3 screws. The longer M2 screws need a Phillips screwdriver.

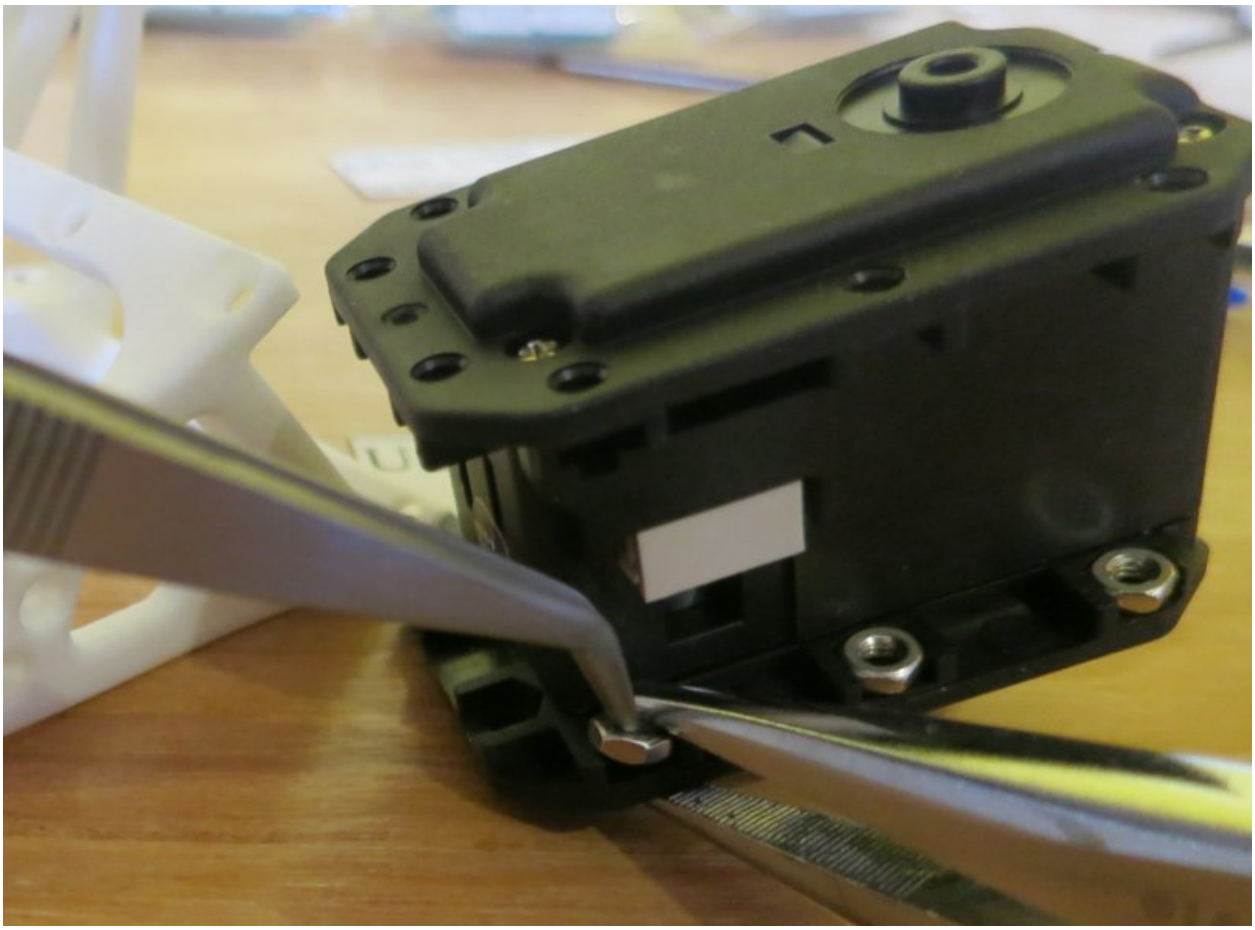
Putting the nuts

To attach structural parts on the body of the servomotors, you have to first insert the nuts in their sites. This step may be quite painful if you don't have elfic fingers (there are less nuts to insert in the AT servomotors than in the T version used for the videos).

Here's my tip: take the nut using thin tweezers and bring it in the site with the right orientation. Put the end of the tweezers in the hole to ensure good alignment. Then use flat pincers to adjust the nut.







These nuts correspond to diameter 2.5mm screws, Allen wrench 2mm.

To build a full Poppy Humanoid robot, an electrical screwdriver is strongly advised!

Addressing Dynamixel motors

By default, every Dynamixel servomotor has its ID set to 1. To use several servomotors in a serial way, each of them must have a unique ID.

Using your robot's main board and Ipython notebooks

This is the prefered solution, but it implies that you have a ready-to-use Odroid or Raspberry Pi board, connected to the network, with ipython notebook server started (or a webapp to start the server).

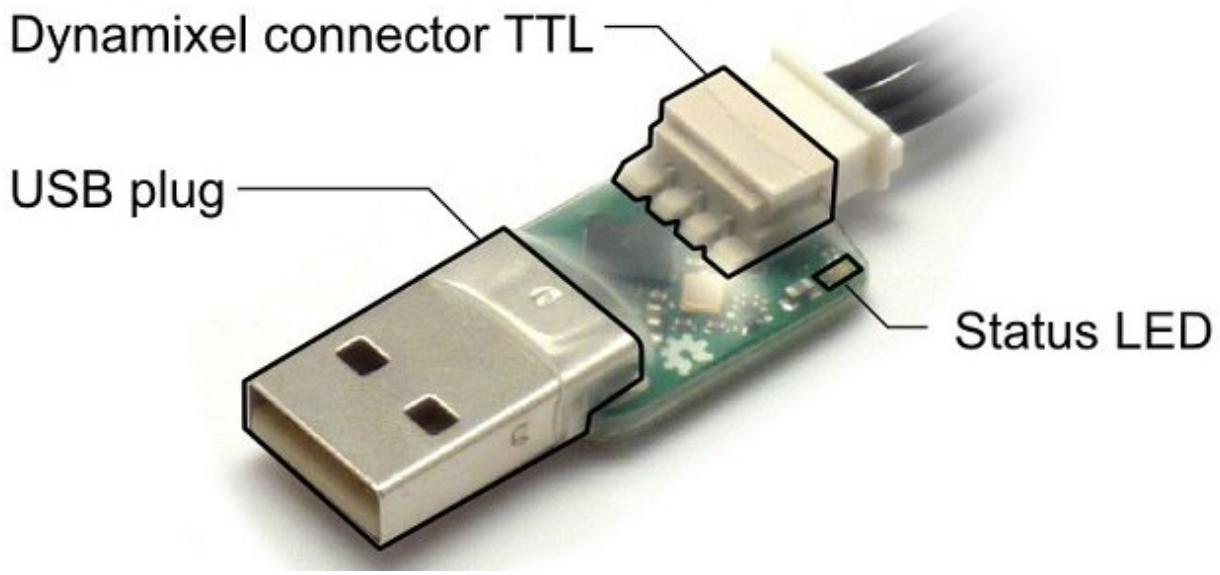
Connect to your robot's jupyter server by entering the URL poppy.local:8888

Installing the driver for USB2AX

There are two devices allowing you to connect your Dynamixel bus to your computer: USB2Dynamixel and USB2AX.

The first one is created by Robotis (the conceptors of the Dynamixel devices) and can be used to control RS-232(serial), RS-485 (4-pin) and TTL (3-pin) busses. Be sure to set the selector in the position corresponding to the protocol you want to use. More info

USB2AX is a miniaturized version of the USB2Dynamixel able to control only TTL busses. We are using USB2AX to the rest of the doc.



- Due to differences in sensibilities, new MX-28 and MX-64 servos communicate at a 57600 baudrate with USB2AX and 57142 for USB2Dynamixel.*

USB2AX is the device that will connect the Poppy Humanoid robot's head to the Dynamixel servomotors. It can also be used to control the servomotors directly from your computer and that's what we will do to address the motors.

On Linux, no installation is needed, but you must add yourself in the group which own the USB serial ports. It is "dialout" or "uucp" depending on your distribution:

```
sudo addgroup $USER dialout
sudo addgroup $USER uucp
```

Otherwise, the driver is available here.

Don't forget to power up your motors (using a SMPS2Dynamixel) otherwise they won't be detected !

Installing the scanning software

Use one of the two following software to access the Dynamixel servomotors registers:

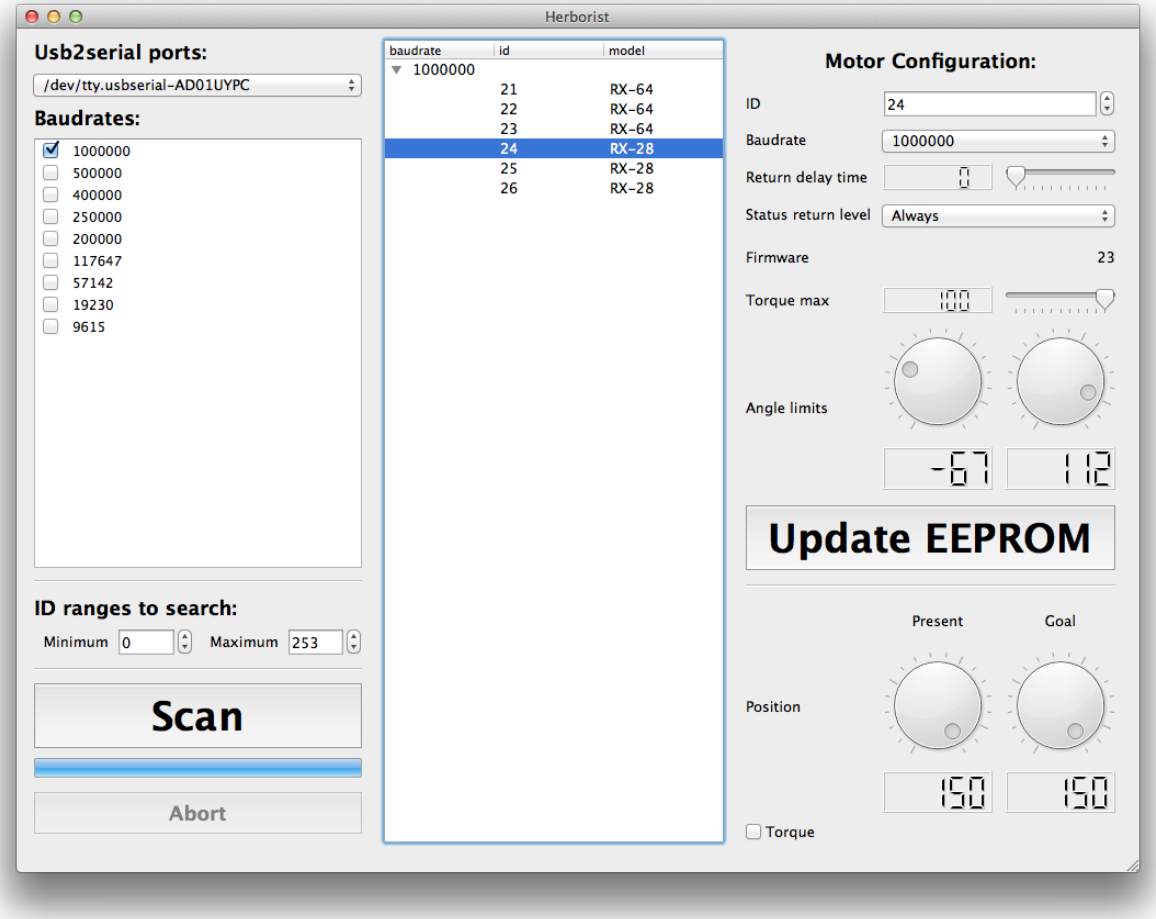
- Herborist: tool created by the Poppy Project team.
- Dynamixel Wizard: windows-only tool provided by Robotis.

Herborist comes with the Pypot library, but needs the additional library PyQt4 for graphical interface (sudo may not be needed).

```
sudo apt-get install python-qt4 python-numpy python-scipy python-pip
sudo pip install pypot
```

It should then be directly accessible in a terminal:

```
herborist
```



Connect each motor **one by one** to the USB2AX and use the 'scan' button in Herborist or Dynamixel Wizard to detect it. If it's a new motor, it should have ID 1 and baudrate 57600bps, apart from AX-12A servos who already have a 1000000 baudrate.

You have to set:

- ID corresponding to the naming convention
- Baudrate to 1 000 000 bps
- Return delay time to 0 ms instead of 0.5 ms

In Herborist, don't forget to click on the 'Update EEPROM' button so the changes are taken in account.

Naming conventions

If you want your software object to correspond to your robot without having to modify the configuration file, you should stick to the robot naming and addressing convention, described in the assembly doc and in each robot's library. This will ensure that, in your code, when you use a motor's name, you will really send orders to the corresponding physical motor.

2.1.2 Assembly guide for a Poppy Humanoid or Poppy Torso robot

Bill Of Material - - Poppy Humanoid 1.0

3D printing

Last full release: https://github.com/poppy-project/Poppy-Humanoid/releases/

lower_limbs:

- 2x leg *(white polished polyamid)*
- 1x thigh_right *(white polished polyamid)*
- 1x thigh_left *(white polished polyamid)*
- 1x hip_right *(white polished polyamid)*
- 1x hip_left *(white polished polyamid)*
- 1x pelvis *(white polished polyamid)*
- 1x simple_foot_left *(white polished polyamid)*
- 1x simple_foot_right *(white polished polyamid)*
- 2 x hip_z_to_hip_y-connector *(white polished polyamid)*

torso

- 2x double_rotation_MX64_link *(white polyamid)*
- 1x i101-Set_to_MX64_link *(white polyamid)*
- 1x abdomen *(white polished polyamid)*
- 1x spine *(white polished polyamid)*
- 2x double_rotation_MX28_link *(white polyamid)*
- 1x i101-Set_to_MX28_link *(white polyamid)*
- 1x chest *(white polished polyamid)*

upper_limbs

- 1x shoulder_right *(white polished polyamid)*
- 1x shoulder_left *(white polished polyamid)*
- 2x arm_connector *(white polished polyamid)*
- 2x upper_arm *(white polished polyamid)*

- 1x forearm_left *(white polished polyamid)*
- 1x forearm_right *(white polished polyamid)*
- 1x hand_right *(white polished polyamid)*
- 1x hand_left *(white polished polyamid)*

head

- 1x neck *(white polished polyamid)*
- 1x head-back *(white polished polyamid)*
- 1x head-face *(white polished polyamid)*
- 1x support_camera *(white polyamid)*
- 1x screen *(resine transparent)*
- 1x hide-screen *(black polyamid)*
- 1x speaker_left *(black polyamid)*
- 1x speaker_right *(black polyamid)*
- 1x fake_manga_screen *(black polyamid)*

Electronics

Embedded control

- 2x USB2AX
- Hardkernel Odroid U3
- 8GB eMMC Module U Linux
- Cooling Fan U3 (Optionnal)
- USB hub: here ou here

Power Supply

- DC Plug Cable Assembly 2.5mm
- 5V DC convertor

Audio/video

- 2x speakers
- ampli stereo

Communication

- Nano Wifi Dongle

Sensors

- Camera Videw **with FOV 120° or 170°!!** + USB cable
- Sparkfun Razor 9DoF IMU (Optionnal)
- Manga Screen (Optionnal)

Robotis

Actuators Dynamixel

- 19 x MX-28AT
- 4 x MX-64AT
- 2 x AX-12A

Parts

- 19x HN07-N101 set
- 12x HN07-i101 Set
- 4x HN05-N102 Set
- 4x HN05-i101 Set

Visserie:

- 1x Wrench Bolt M2*3 (200 pcs)
- 1x Wrench Bolt M2.5*4 (200 pcs)
- 1x Wrench Bolt M2.5*6 (200 pcs)
- 1x Wrench Bolt M2.5*8 (200 pcs)
- 1x BIOLOID Bolt Nut Set BNS-10
- 1x Nut M2.5 (400 pcs)
- 1x N1 Nut M2 (400 pcs)

Cables

- 3x SMPS2Dynamixel
- 1x SMPS 12V 5A PS-10
- 3x BIOLOID 3P Extension PCB
- 1x Robot Cable-3P 60mm 10pcs
- 1x Robot Cable-3P 100mm 10pcs
- 1x Robot Cable-3P 140mm 10pcs
- 1x Robot Cable-3P 200mm 10pcs

Custom:

- 3x cable-3P 22cm
- 3x cable-3P 25cm
- 2x cable-3P 50cm
- 2x Cable-4P 200mm avec fils D+/D- coupés

Tools

- 1x http://www.leroymerlin.fr/v3/p/produits/lot-de-6-mini-pinces-dexter-e148011
- 1x http://www.leroymerlin.fr/v3/p/produits/set-de-micro-vissage-de-precision-mixte-dexter-e140690
- 1x http://www.leroymerlin.fr/v3/p/produits/set-de-micro-vissage-de-precision-mixte-tivoly-11501570026-e59080
- 2x frein fillet http://fr.farnell.com/jsp/search/productdetail.jsp?SKU=1370152
- 3x clé allen 1.5 mm
- 2x clé allen 2 mm
- 1x clé allen 2.5mm
- scotch blanc http://fr.farnell.com/jsp/search/productdetail.jsp?SKU=1825466

Bill Of Material - - Poppy Torso 1.0

3D printing

torso

- 1x support_for_table *(white polished polyamid)*
- 1x spine *(white polished polyamid)*
- 2x double_rotation_MX28_link *(white polyamid)*
- 1x i101-Set_to_MX28_link *(white polyamid)*
- 1x chest *(white polished polyamid)*

upper_limbs

- 1x shoulder_right *(white polished polyamid)*
- 1x shoulder_left *(white polished polyamid)*
- 2x arm_connector *(white polished polyamid)*
- 2x upper_arm *(white polished polyamid)*
- 1x forearm_left *(white polished polyamid)*
- 1x forearm_right *(white polished polyamid)*
- 1x hand_right *(white polished polyamid)*
- 1x hand_left *(white polished polyamid)*

head

- 1x neck *(white polished polyamid)*
- 1x head-back *(white polished polyamid)*
- 1x head-face *(white polished polyamid)*
- 1x support_camera *(white polyamid)*
- 1x screen *(resine transparent)*
- 1x hide-screen *(black polyamid)*
- 1x speaker_left *(black polyamid)*
- 1x speaker_right *(black polyamid)*
- 1x fake_manga_screen *(black polyamid)*

Electronics

Embedded control

- 1x USB2AX
- Hardkernel Odroid U3
- 8GB eMMC Module U Linux
- Cooling Fan U3 (Optionnal)
- USB hub: here ou here

Power Supply

- DC Plug Cable Assembly 2.5mm
- Robotis SMPS 12V 5A PS-10
- 5V DC convertor

Audio/video

- 2x speakers
- ampli stereo

Communication

- Nano Wifi Dongle

Sensors

- Camera Videw **with FOV 120° or 170°!!** + USB cable
- Sparkfun Razor 9DoF IMU (Optionnal)
- Manga Screen (Optionnal)

Custom TODO

Robotis

Dynamixel

- 11 x MX-28AT
- 2 x AX-12A

Parts

- 11x HN07-N101 set
- 6x HN07-i101 Set

Visserie:

- 1x Wrench Bolt M2*3 (200 pcs)
- 1x Wrench Bolt M2.5*4 (200 pcs)
- 1x Wrench Bolt M2.5*6 (200 pcs)
- 1x Wrench Bolt M2.5*8 (200 pcs)
- 1x BIOLOID Bolt Nut Set BNS-10
- 1x Nut M2.5 (400 pcs)
- 1x N1 Nut M2 (400 pcs)

Cables

- 1x SMPS2Dynamixel
- 2x BIOLOID 3P Extension PCB
- 1x Robot Cable-3P 60mm 10pcs
- 1x Robot Cable-3P 100mm 10pcs
- 1x Robot Cable-3P 140mm 10pcs
- 1x Robot Cable-3P 200mm 10pcs

Custom:

- 3x cable-3P 22cm
- 3x cable-3P 25cm
- 2x cable-3P 50cm
- 2x Cable-4P 200mm

Other

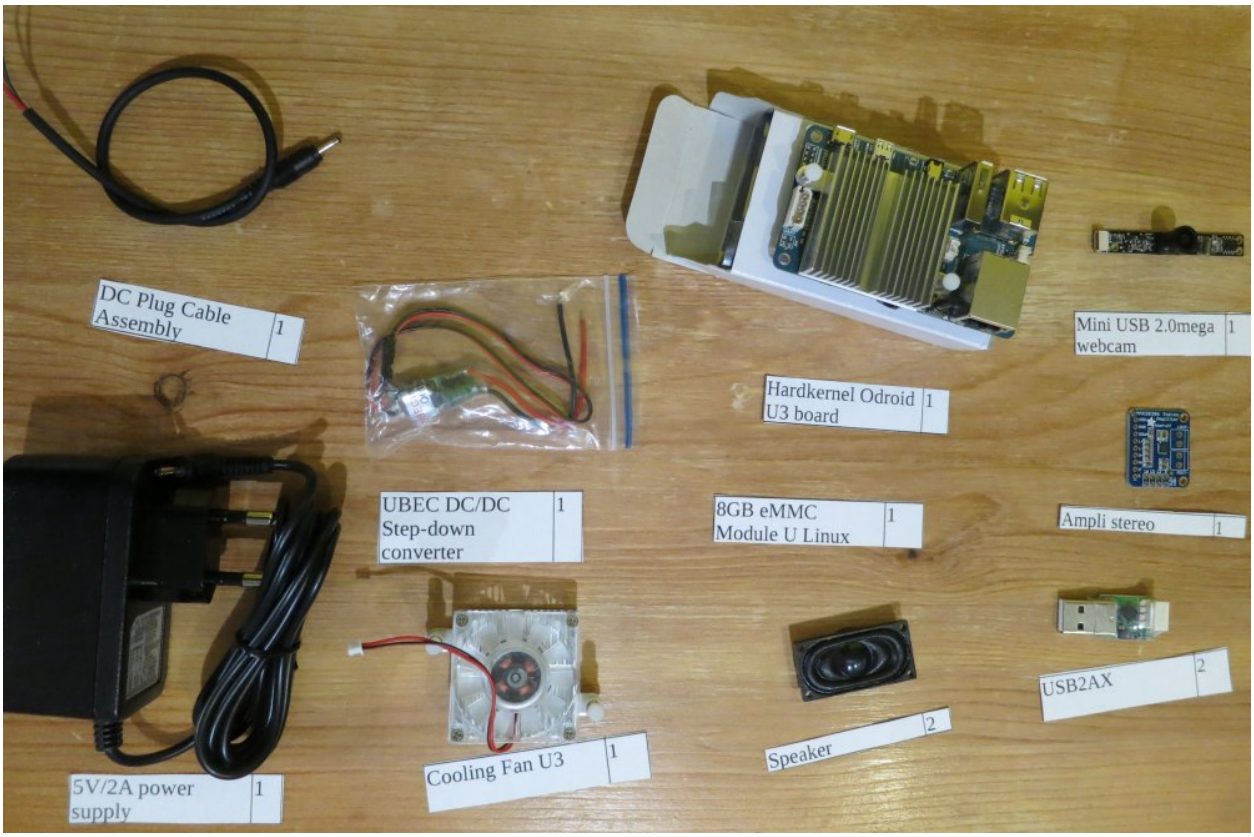
- 100mm with lever arm suction pad
- 2x M5 nuts
- 2X M5x20mm screws

Tools

- 1x http://www.leroymerlin.fr/v3/p/produits/lot-de-6-mini-pinces-dexter-e148011
- 1x http://www.leroymerlin.fr/v3/p/produits/set-de-micro-vissage-de-precision-mixte-dexter-e140690
- 1x http://www.leroymerlin.fr/v3/p/produits/set-de-micro-vissage-de-precision-mixte-tivoly-11501570026-e59080
- 2x frein fillet http://fr.farnell.com/jsp/search/productdetail.jsp?SKU=1370152
- 3x clé allen 1.5 mm
- 2x clé allen 2 mm
- 1x clé allen 2.5mm
- scotch blanc http://fr.farnell.com/jsp/search/productdetail.jsp?SKU=1825466

Head assembly

\



\

| Sub-assembly name | Motor name | Type | ID | |———————-|:———:|:——-:|:–:| | Head | head_y | AX-12A | 37 |

Setup of the Odroid board

The Odroid is normally shipped with a eMMC module with Ubuntu 1.14 already flashed (it should have a red sticker on it). Simply plug it on the Odroid board and power it. After boot time, it should have the red light steady and the blue light flashing.

If you don't have a pre-flashed eMMC module, follow these instructions: https://github.com/poppy-project/poppy_install

Connect the Odroid board to your network using an ethernet cable. You have to access a wired network for initial setup (I tried link-local without success).

Windows users may wish to install the Bonjour software (the link is for the printer version, which does very well what we want it to do). Bonjour is installed by default on Linux and Mac. It is used to communicate with another device using its name instead of its IP.

You should get an answer if you type:

```
ping odroid.local
```

Windows users now probably wish to install Putty or any SSH client. Linux and Mac users have one installed by default. Then:

```
ssh odroid@odroid.local
```

Password is odroid. Congratulations, you are now inside the Odroid!

Make sure the Odroid board has access to the internet and enter:

```
curl -L https://raw.githubusercontent.com/poppy-project/
poppy_install/master/poppy_setup.sh | sudo bash
```

Enter the odroid password. This command will download and run a script which will download and prepare installation. The board asks for a reboot:

```
sudo reboot
```

You loose the SSH connection. The board has changed hostname and password, so wait for the blue light to flash regularly and connect with:

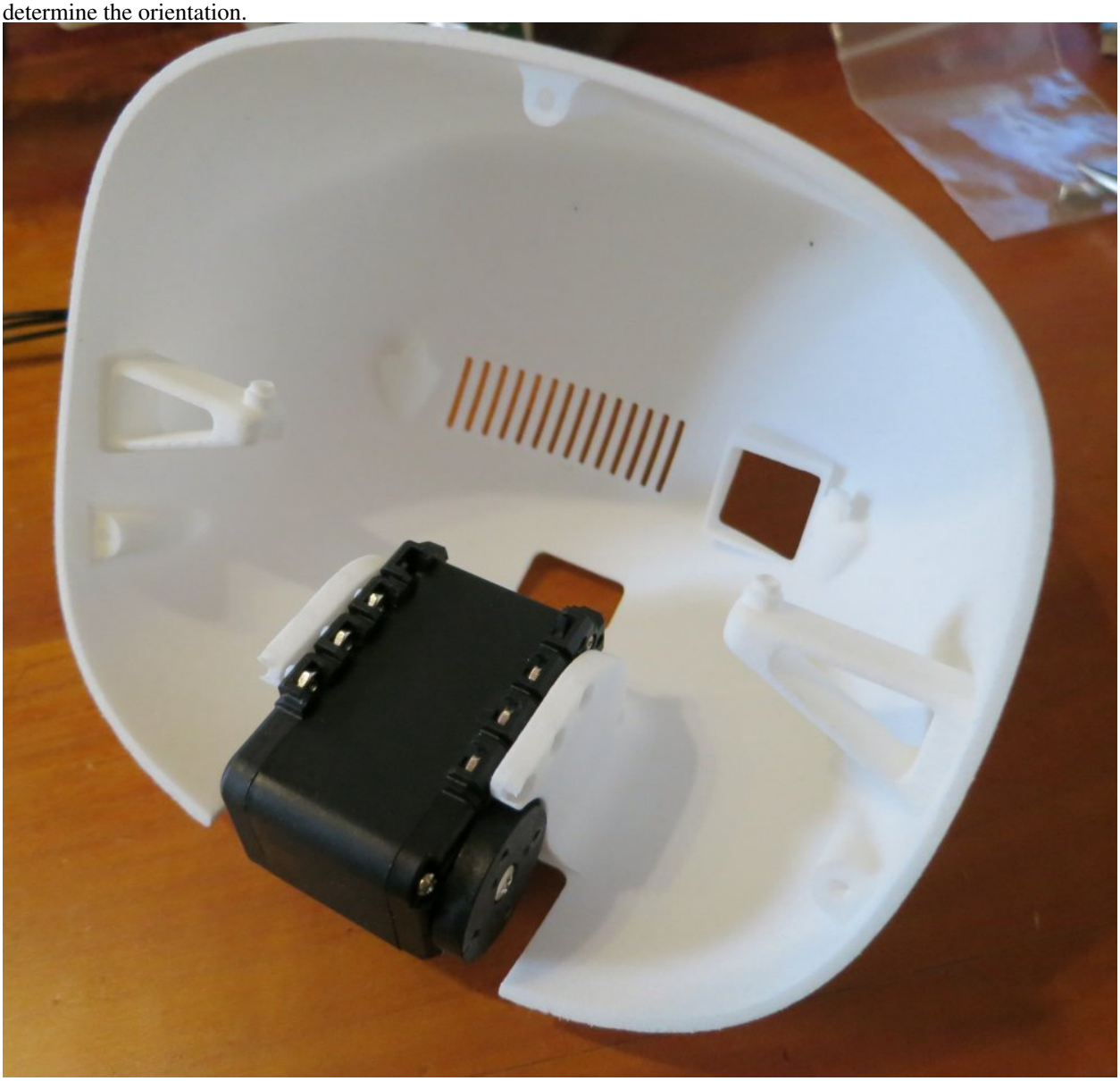
```
ssh poppy@poppy.local
```

As you guessed, password is poppy. Installation process takes place automatically (and takes a while). When you see 'System install complete', do a Ctrl+C to finish. After a new reboot, your Odroid board is ready.

Neck assembly

The last servomotor is head_y, a AX-12A. Set its ID to 37 and response time to 0 (baudrate is already at 1000000).

Screw the neck to head_z servo (M2x8mm screws). There are marks on the neck and on the servo to help you determine the orientation.



Put 2 nuts in the servo case and attach it in the head_back part.

\

Assemble the servo on the neck (2 screws on the controlled side, the big screw on the other side). You again have marks on the neck and on the servo for orientation.

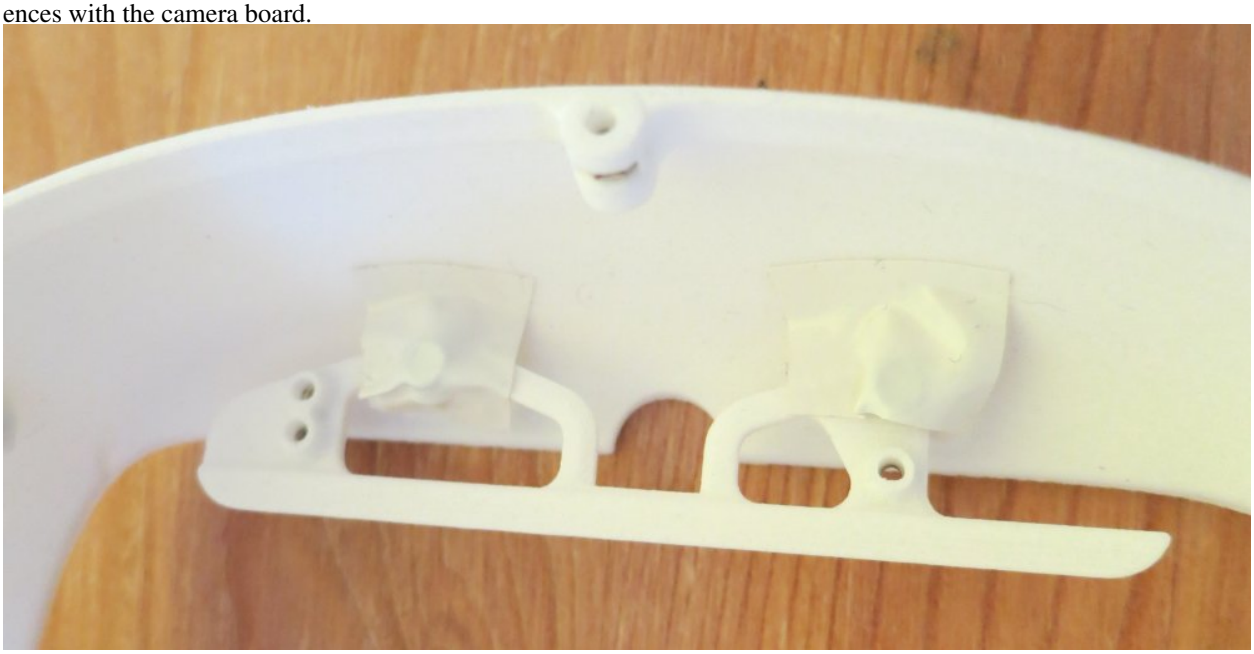
Connect head_y to the dispatcher by passing the cable through the hole in the head.

Plug a 500mm cable from the pelvis SMPS2Dynamixel board to the back of the head. Attach a USB2AX at the end of this cable in the head.

Use a 140mm cable to connect the head_y motor to another USB2AX.

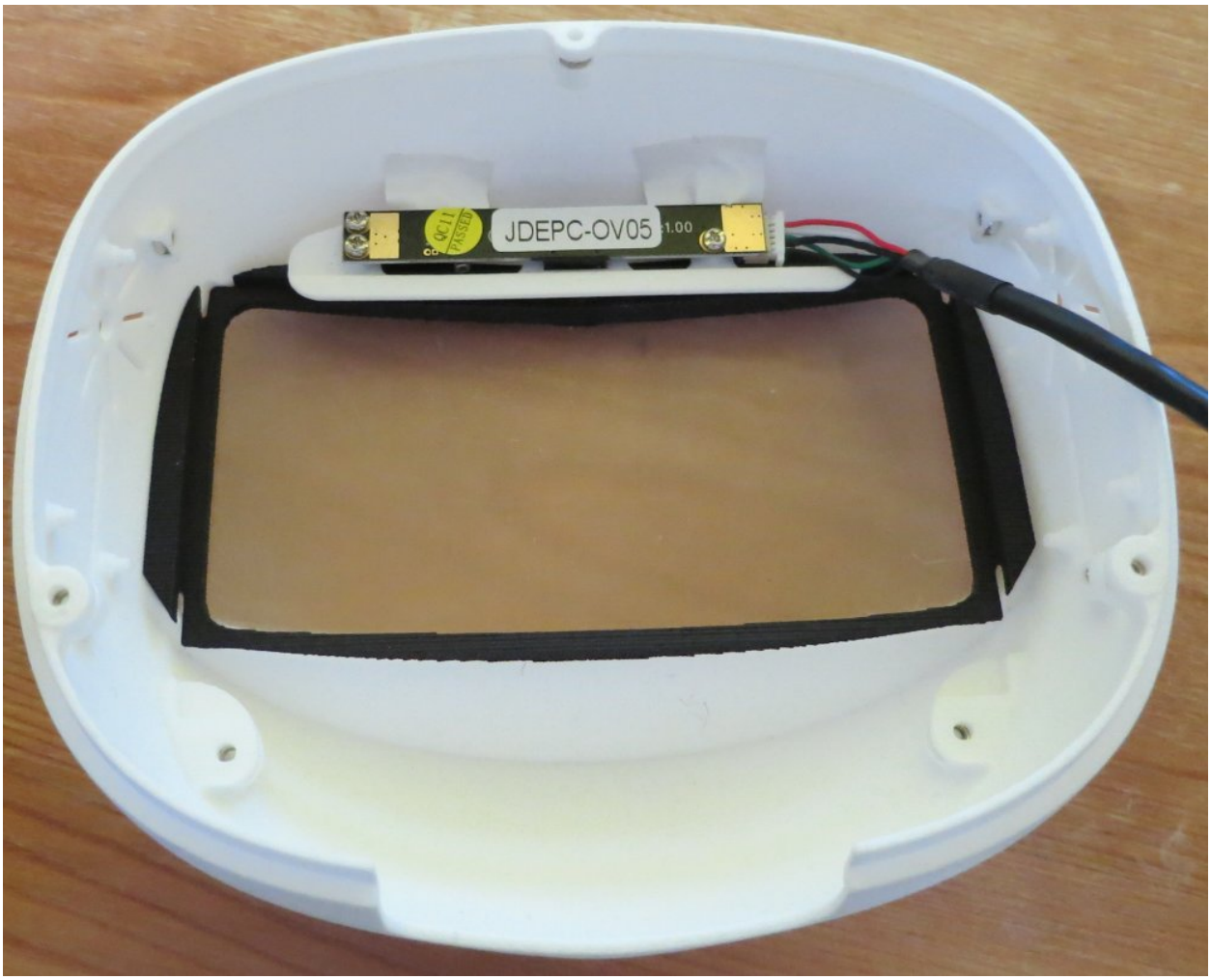
Camera and screen

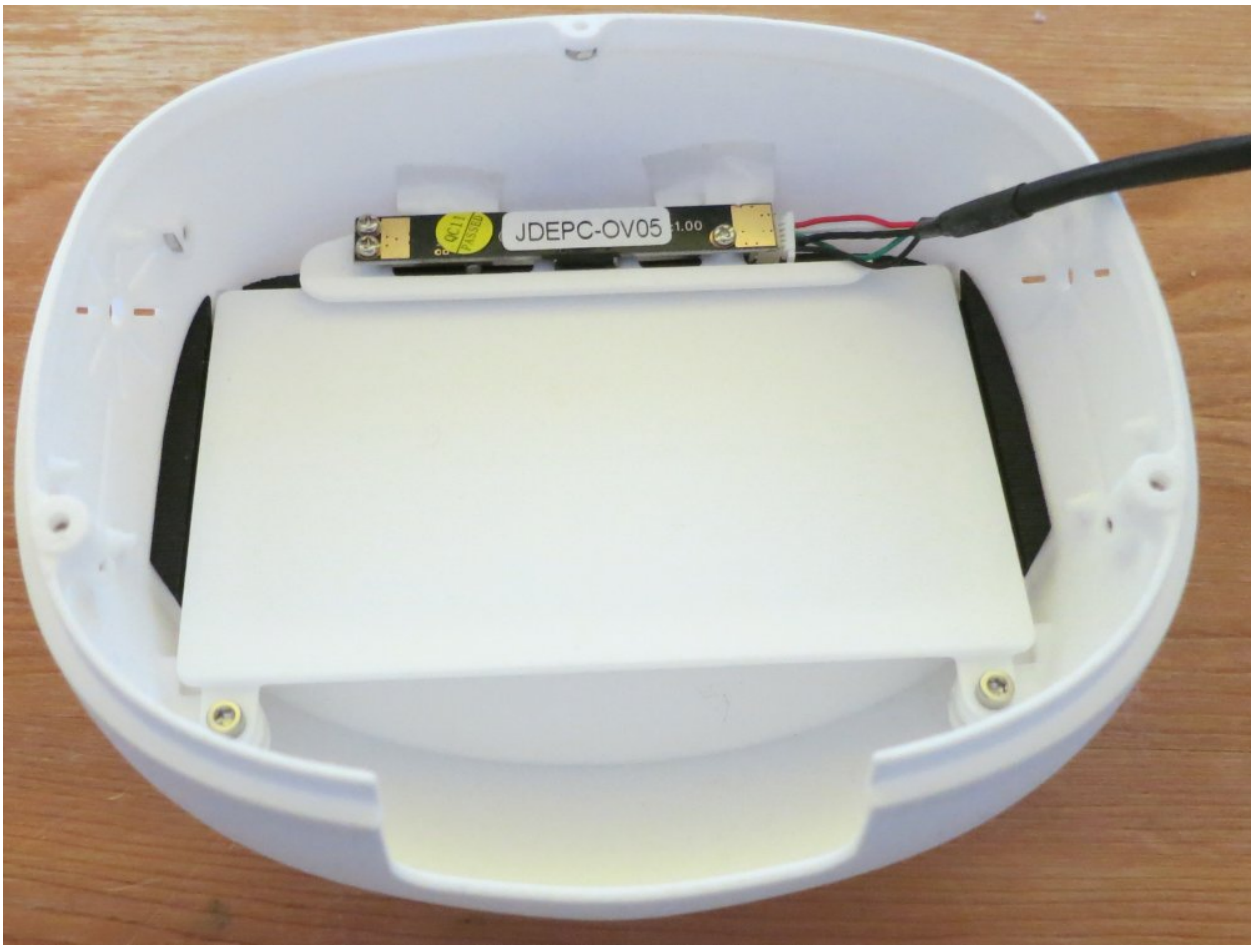
Attach the camera support to head_front using M2.5x4mm screws. Put tape on the screws to avoid electrical interferences with the camera board.



Attach the camera to its support using 3 M2x6mm screws.

Put the screen and screen cover in the head. Attach the manga screen (or the fake one) with 2 M2.5x6mm screws.

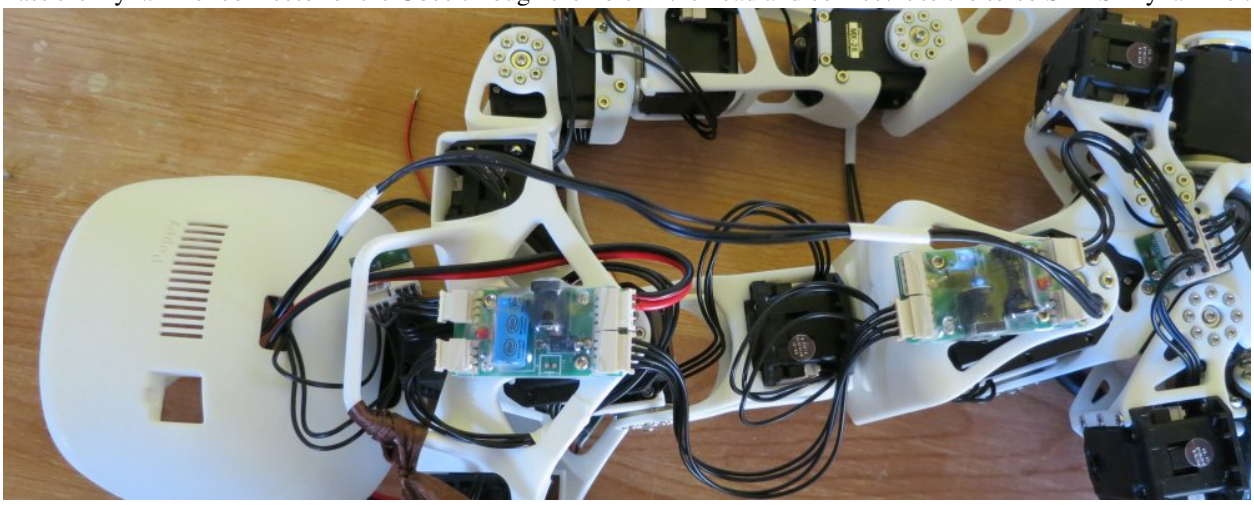

JDEPC-OV05



Electronics

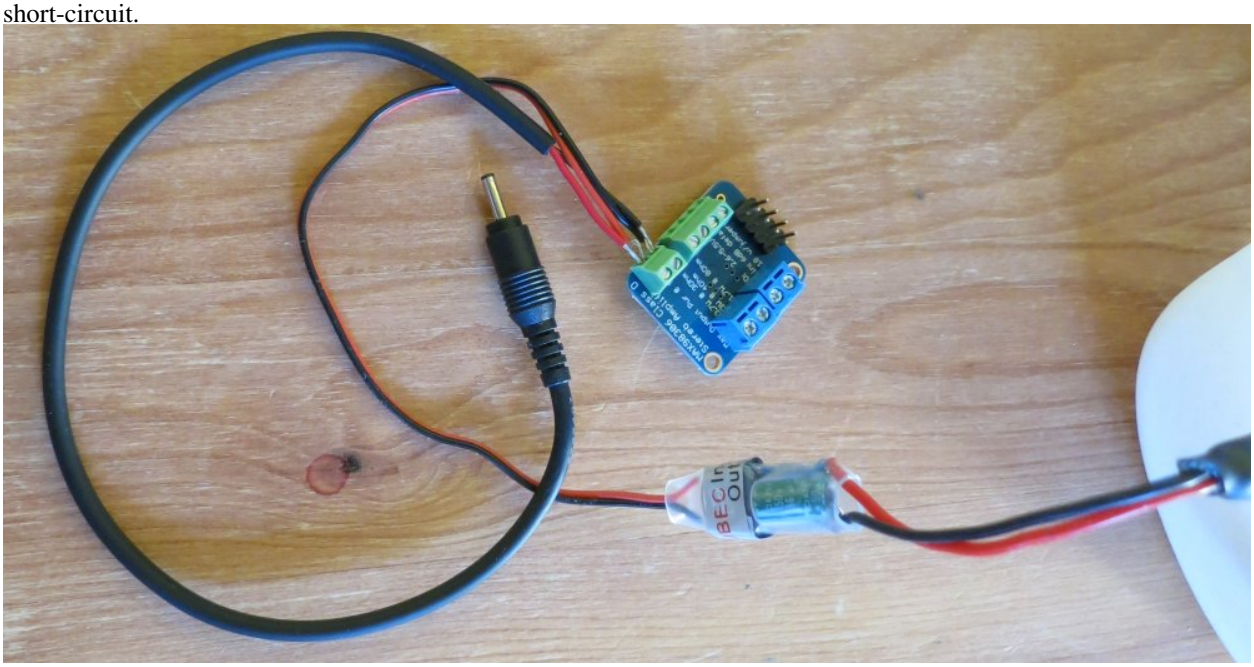
If you don't have pre-soldered components, see: https://github.com/poppy-project/Poppy-minimal-head-design/blob/master/doc/poppy_soldering.md

Pass the Dynamixel connector of the Ubec through the hole in the head and connect it to the torso SMPS2Dynamixel.

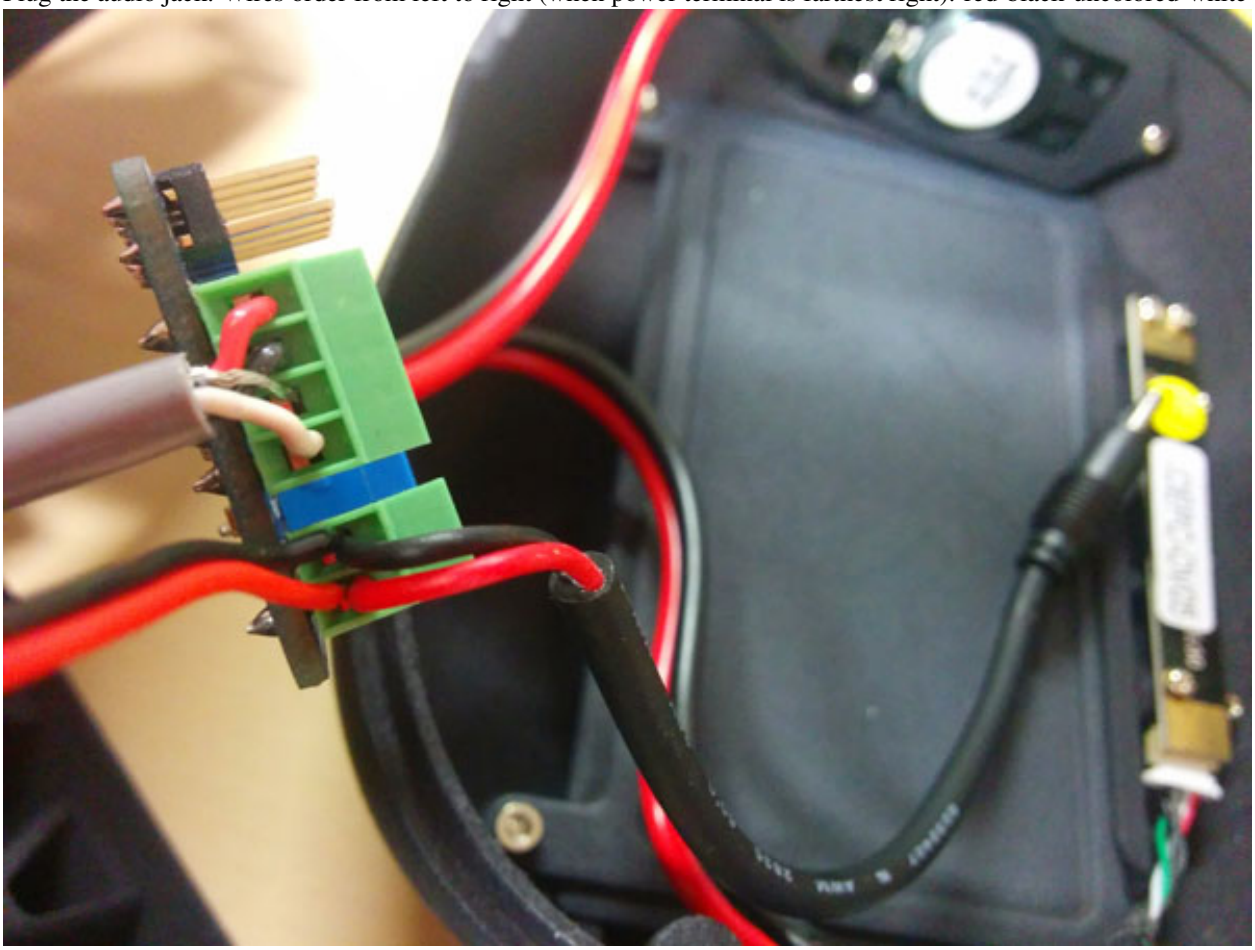


Attach both the other side of Ubec and the Odroid power cable to the audio amplifier. Be sure no to allow any

short-circuit.

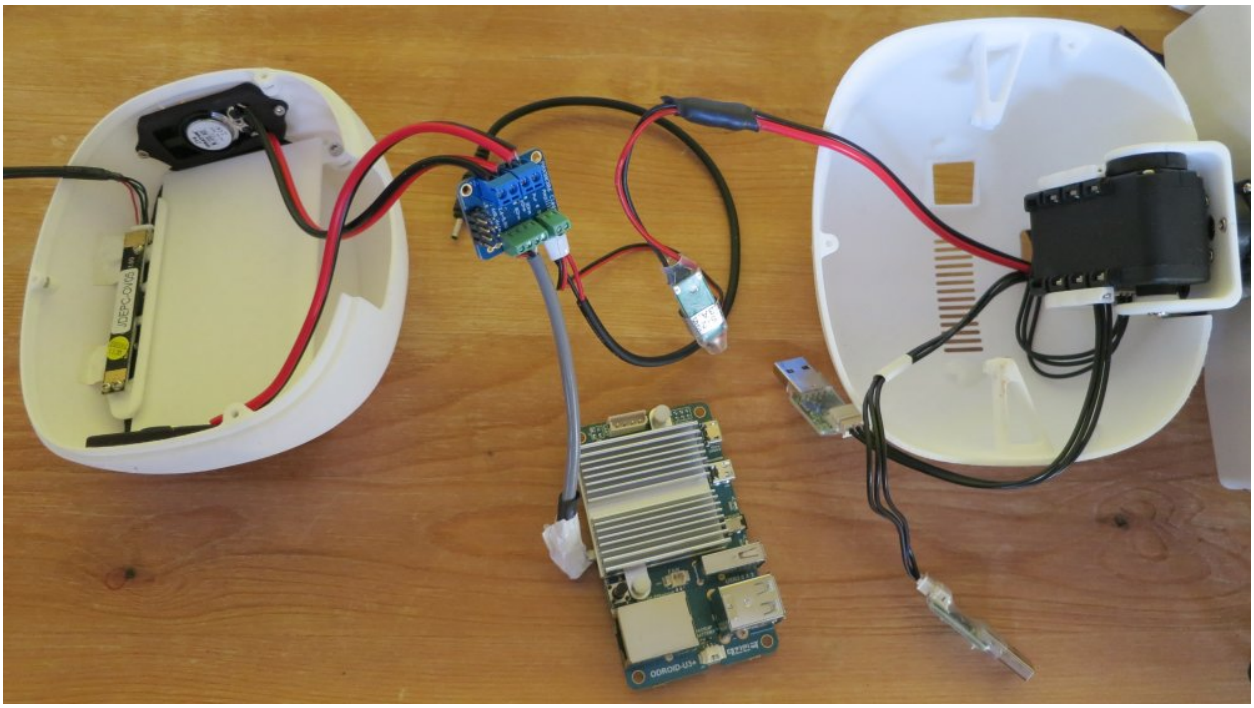


Plug the audio jack. Wires order from left to right (when power terminal is farthest right): red-black-uncolored-white

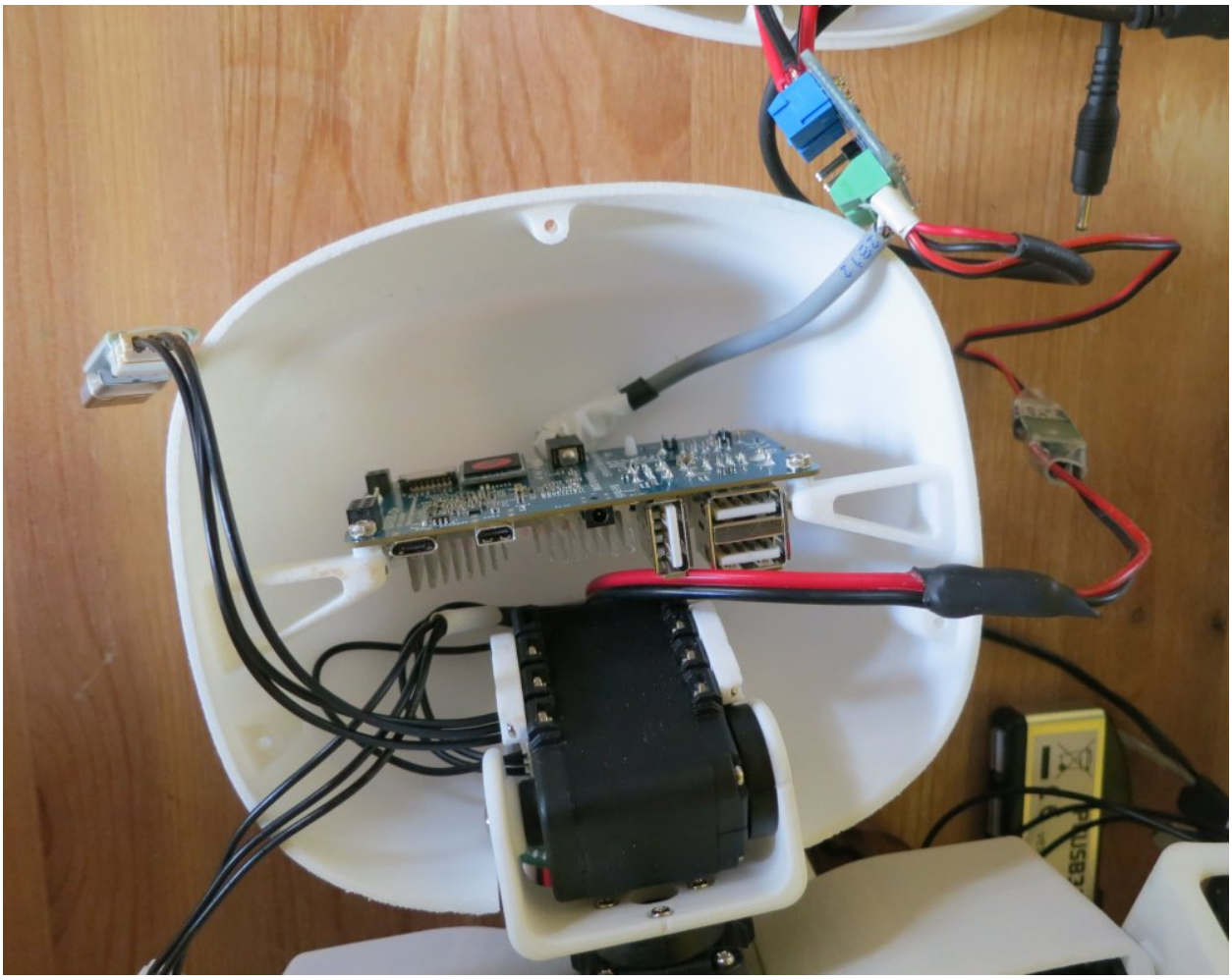


Put 2 nuts around the flowers openings then attach the speakers using M2 x3mm screws.

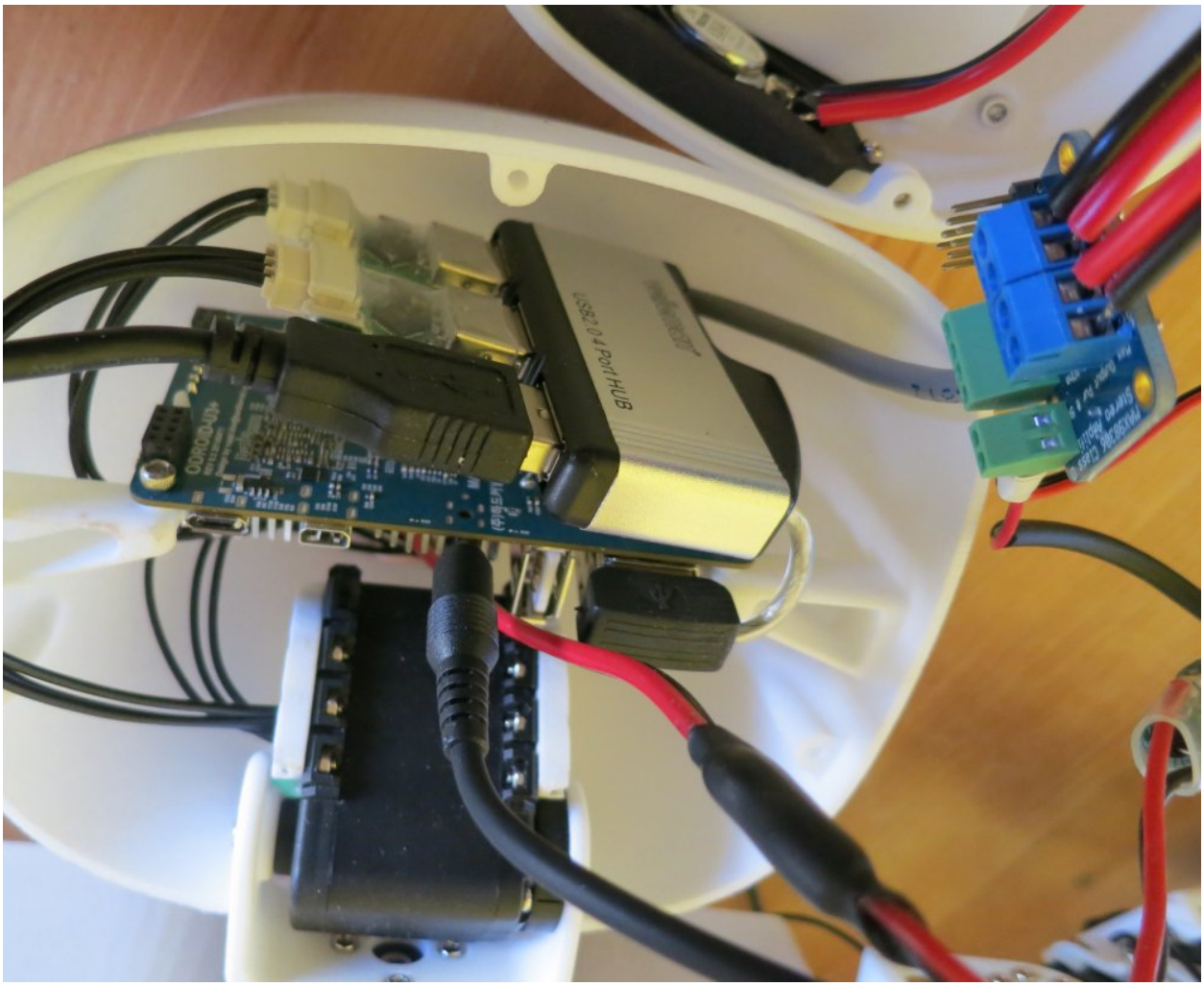
Connect the speakers to audio ampli, left speaker black wire on Lout, Right ampli black wire on Rout.



Plug audio jack in Odroid, then use 2 M2.5x8mm screws to attach the Odroid board. Make sure the Ethernet connector is correctly placed in front of the corresponding hole.

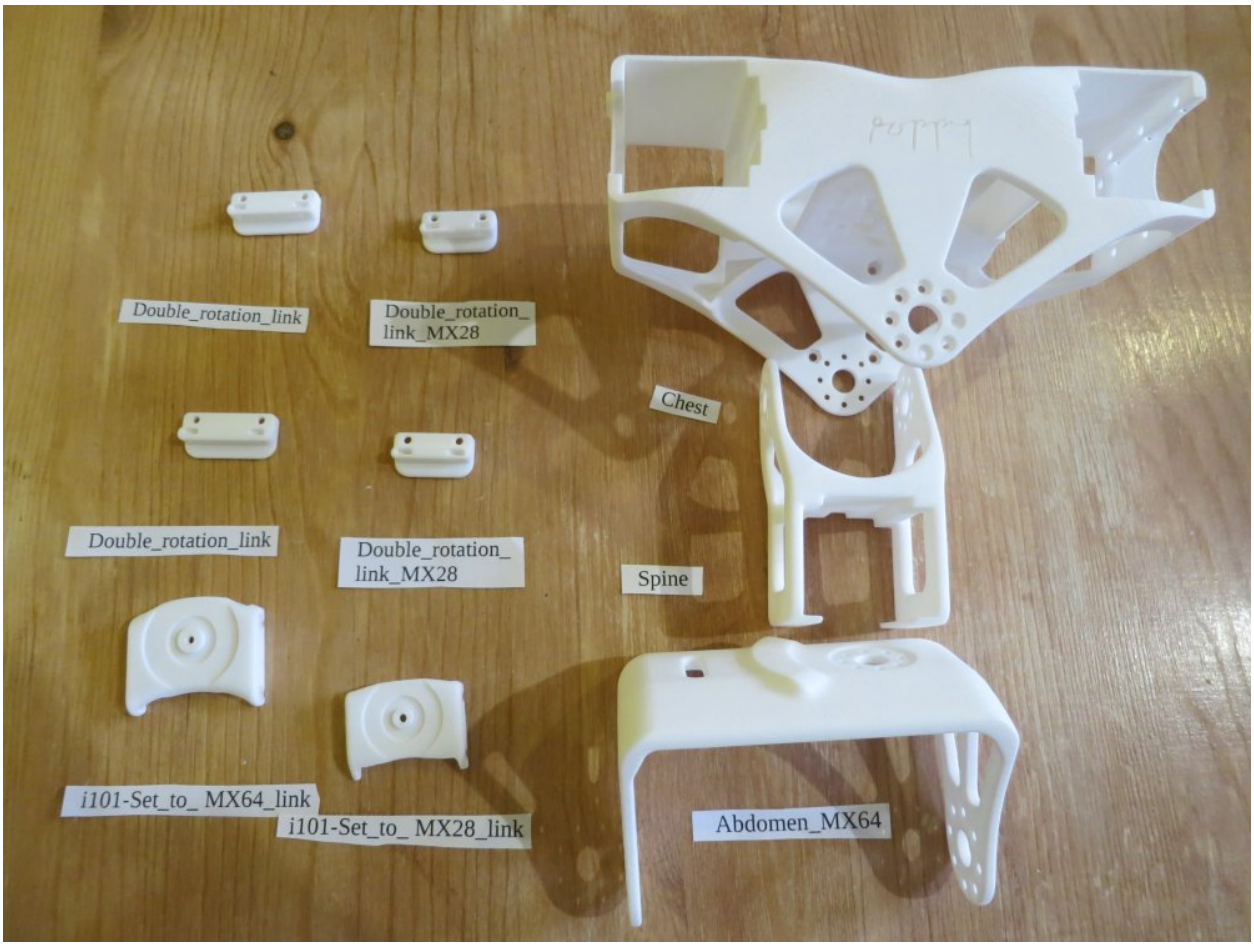


Plug the power jack. On the hub, plug the camera and the two USB2AXs. Plug the wifi dongle and the Razor board if you have them. Push the hub above the Odroid.



Then close the head using 3 M2x8mm screws.

Trunk



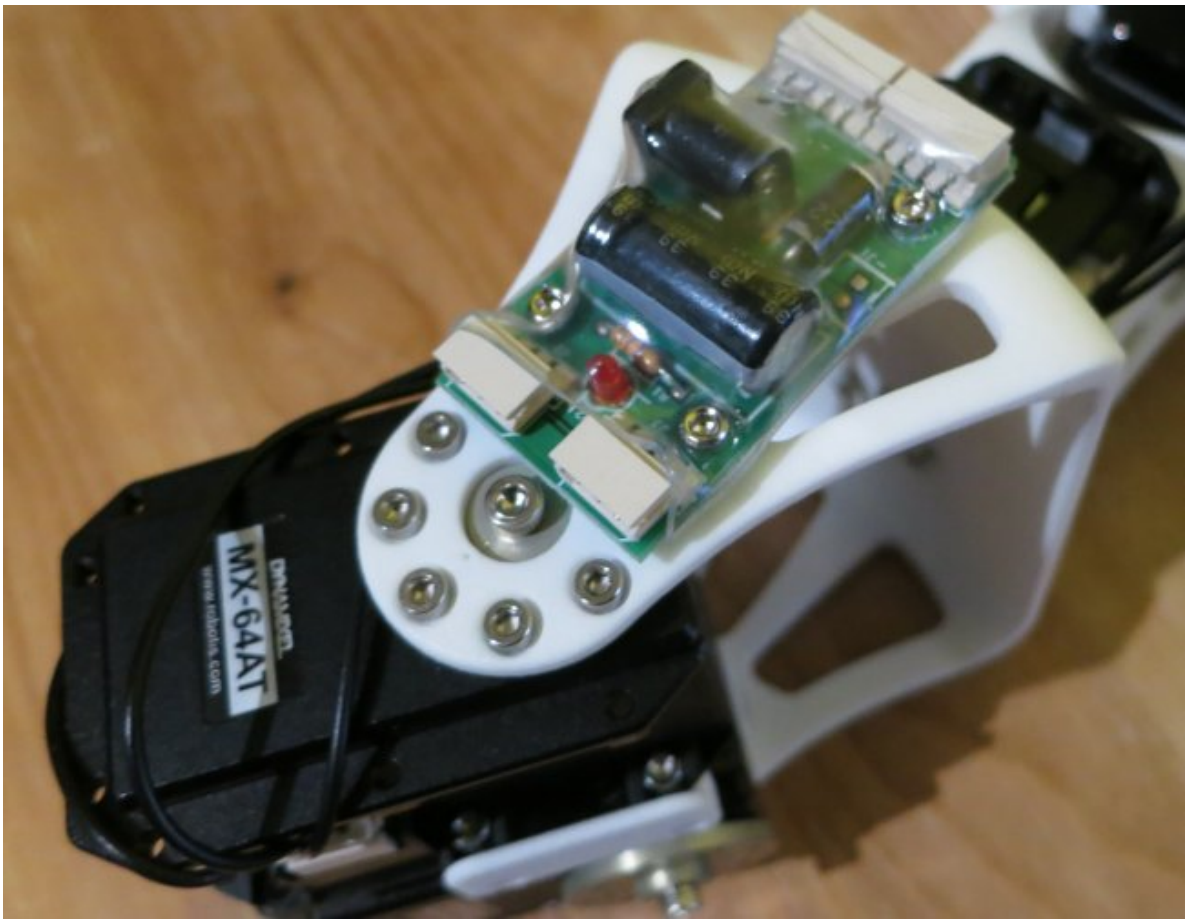
Motors list:

| Sub-assembly name | Motor name | Type | ID | |—————————-|:————:|:———-:|:–:| | Double MX64 | abs_y | MX-64AT | 31 | | Double MX64 | abs_x | MX-64AT | 32 | | Spine | abs_z | MX-28AT | 33 | | Double MX28 | bust_y | MX-28AT | 34 | | Double MX28 | bust_x | MX-28AT | 35 | | Chest | head_z | AX-12A | 36 | | Chest | l_shoulder_y | MX-28AT | 41 | | Chest | r_shoulder_y | MX-28AT | 51 |

Reminder: be careful with orientation while mounting Dynamixel horns

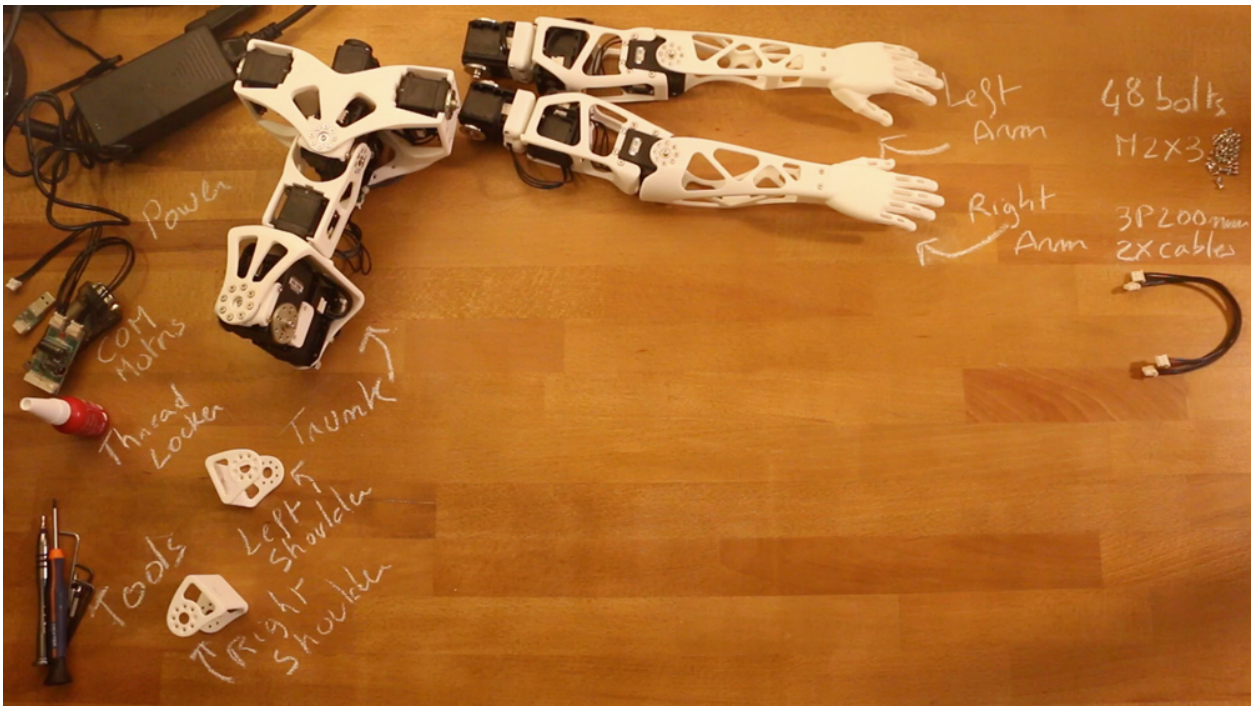
- **Double MX64**
- **Double MX28** Don't screw the i101-Set_to_ MX28_link (the plastic part with a free horn on it) too tightly, or don't screw it at all since you will need to unscrew it during the trunk assembly.
- **Spine**
- **Chest** The video shows a HN07_I101 in the prepared parts, but you don't need it.
- **Trunk assembly** You have to insert the nuts in the chest before mounting the double MX-28 part. You also have to put nuts in the abdomen before mounting the double MX-64 part.

  The abdomen part you have has a "Poppy" mark on the back, while the one on the video don't. You also have holes to screw the SMPS2Dynamiel, instead of sticking it (use 2.5*8mm screws).



Assemble trunk and arms:

- Preparation: 5 min
- Assembly: 15-20 min



Requirements Sub-assemblies:

- Trunk
- Left arm
- Right arm

3D printed parts:

- Left shoulder
- right shoulder

Cables:

- 2x 3P 200mm

Robotis parts:

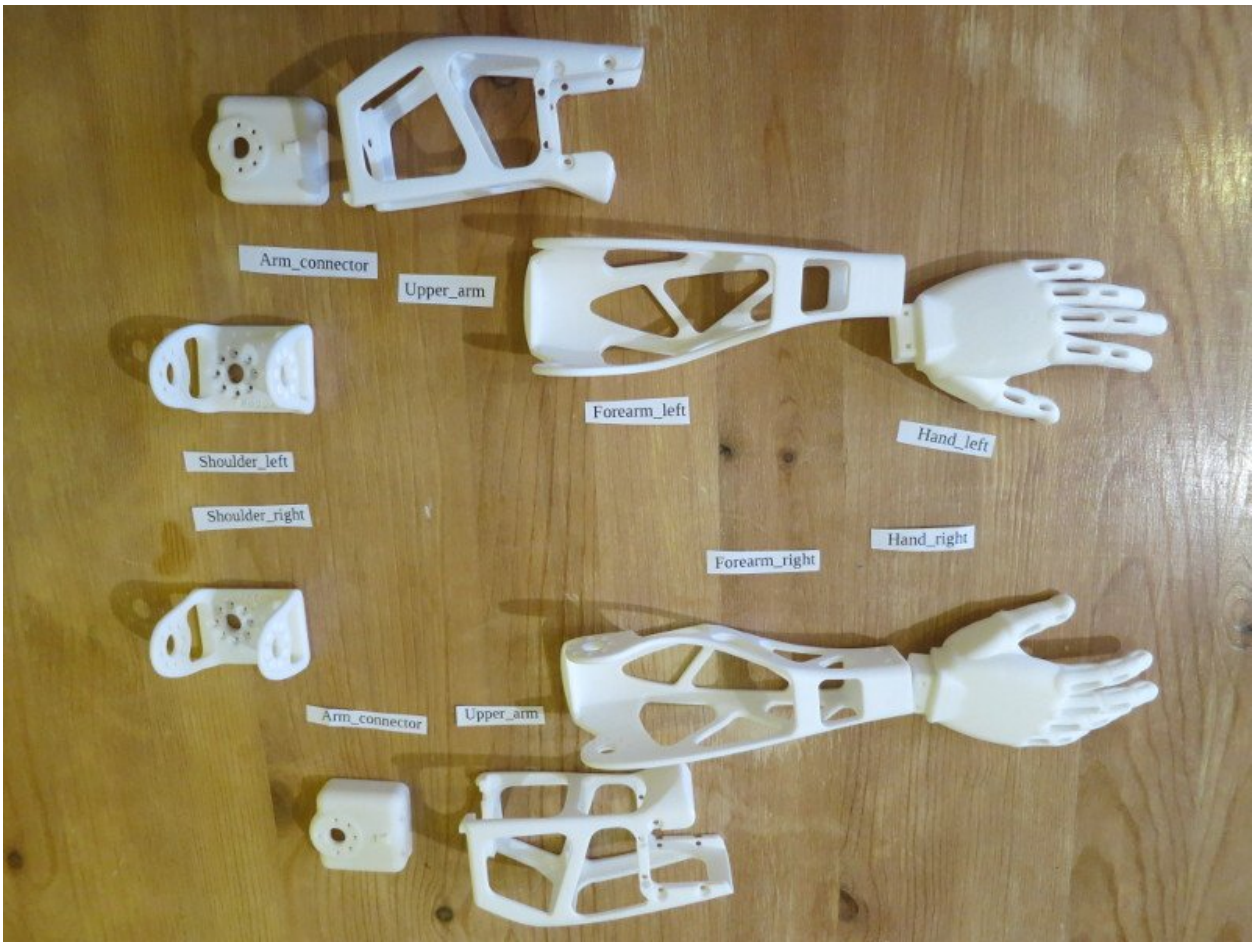
- 48x Bolts M2x3

Motor configuration:

- 1x Alimentation 12V
- 1x SMPS2Dynamixel
- 1x USB2Dynamixel or USB2AX
- A computer...

VIDEO INSTRUCTIONS

Arms assembly



\

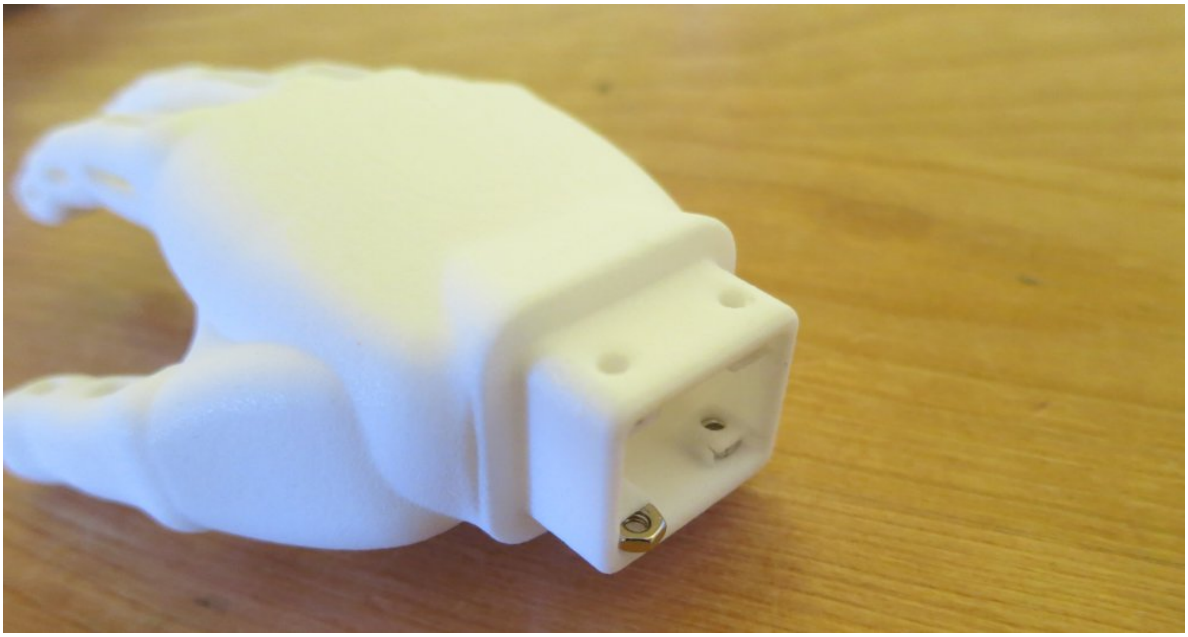
Motors lists:

| Sub-assembly name | Motor name | Type | ID | |———————————- |:————:| |:——:| |:–:| | | Left upper arm/shoulder | l_shoulder_x | MX-28AT | 42 | | Left upper arm | l_arm_z | MX-28AT | 43 | | Left upper arm | l_elbow_y | MX-28AT | 44 |

| Sub-assembly name | Motor name | Type | ID | |———————————|:————:|:——:|:–:| | Right upper arm/shoulder | r_shoulder_x | MX-28AT | 52 | | Right upper arm | r_arm_z | MX-28AT | 53 | | Right upper arm | r_elbow_y | MX-28AT | 54 |

Reminder: be careful with orientation while mounting Dynamixel horns

- **Right/Left forearm** The hand design slightly changed from the videos, but the nuts and screws remain the same.



- **Right/Left upper arm** Plug a 200mm cable in the unused plug before screwing the arm_z motors (ids 43 and 53), because it will be really hard to plug once the motor is inside the structure part.
- **Right/Left upper arm/shoulder**
- **Right/Left arm assembly**
- **Trunk and arms assembly** To distinguish between left and right shoulder parts, look at the three dots: the single dot should be down when the shoulder is in "zero" position (along the shoulder_y motor).

Legs assembly

Steps:

1. Pelvis
2. Left Leg
3. Right Leg
4. Legs assembly
5. Assembly with the torso

3D printed parts required:

Reminder: be careful with orientation while mounting Dynamixel horns

1. Pelvis:

| Sub-assembly name | Motor name | Type | ID | |————————-|:————:|:——-:|:–:| | Pelvis | l_hip_x | MX-28AT | 11 | | Pelvis | r_hip_x | MX-28AT | 21 |

/!\ The instruction shows M2x5mm screws. Use the M2x6mm screws that you can find in the Bolt-nut set BNS-10. ####Pelvis assembly Instructions >>

| Sub-assembly name | Motor name | Type | ID | |————————-|:————:|:——-:|:–:| | Left hip | l_hip_z | MX-28AT | 12 | | Left hip | l_hip_y | MX-64AT | 13 | | Left thigh | l_knee_y | MX-28AT | 14 | | Left shin | l_ankle_y | MX-28AT | 15 |

Sub-assemblies instructions

- **Left Hip**
- **Left Tigh**
- **Left Shin** If you received your Poppy kit from Generation Robots, you can use the custom 220mm cables instead of really short 200mm cables.

Left leg final assembly >>

3. Right leg

| Sub-assembly name | Motor name | Type | ID | |————————-|:————:|:——-:|:–:| | Right hip | r_hip_z | MX-28AT | 22 | | Right hip | r_hip_y | MX-64AT | 23 | | Right thigh | r_knee_y | MX-28AT | 24 | | Right shin | r_ankle_y | MX-28AT | 25 |

Sub-assemblies instructions

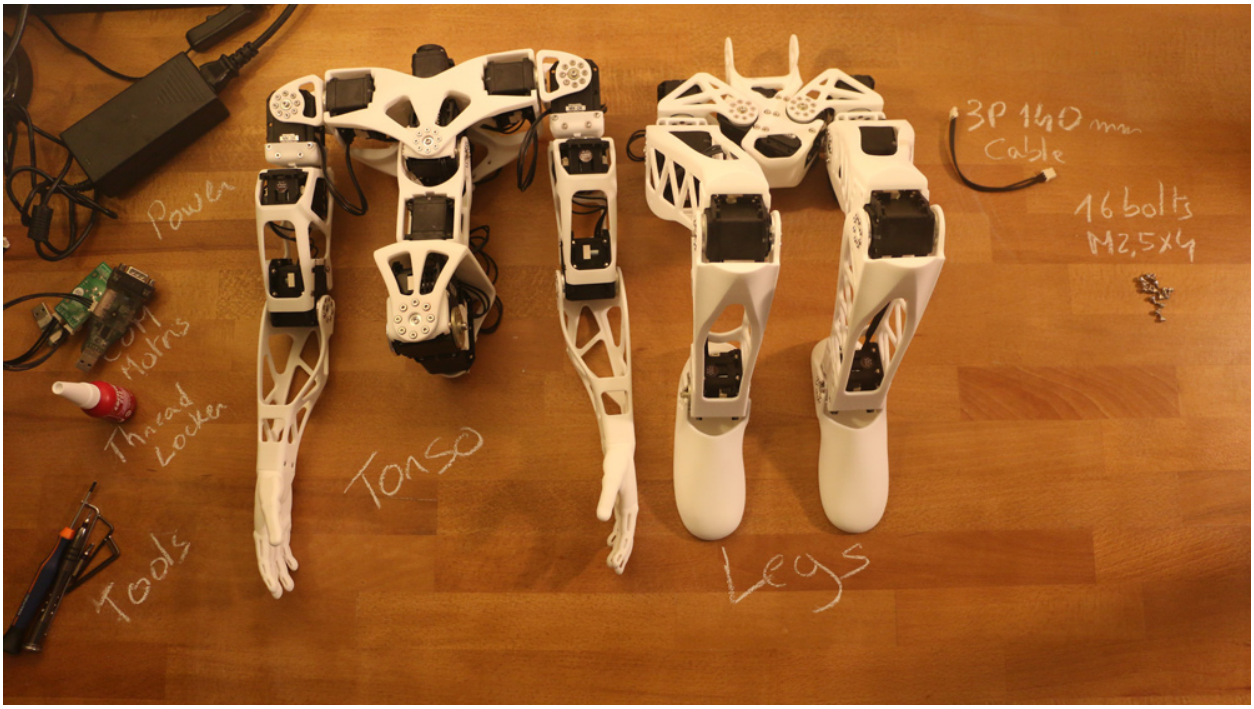
- **Right Hip**
- **Right Tigh**
- **Right Shin** If you received your Poppy kit from Generation Robots, you can use the custom 220mm cables instead of really short 200mm cables.

Right leg final assembly >>

4. Legs/pelvis assembly >>

5. Legs/Torso assembly

- Preparation: 5 min
- Assembly: 5-10 min



Requirement: Sub-assemblies

- Legs
- Torso

Robotis parts:

- 16x Bolts M2.5x4

Cables:

- 1x 3P 140mm

Motor configuration:

- 1x Alimentation 12V
- 1x SMPS2Dynamixel
- 1x USB2Dynamixel or USB2AX
- A computer...

VIDEO INSTRUCTIONS

2.1.3 Assembly guide for Poppy ErgoJr

Introduction

The Poppy Ergo Jr robot is a small and low cost 6-degree-of-freedom robot arm. It consists of very simple forms to be able to print in 3D as easily as possible. The engines have the same functionality as other Poppy creatures but are slightly less powerful and less precise. The advantage being that are also less expensive. The electronic card is not integrated, everything is visible, which is very advantageous to manipulate and change. Its end can be easily modified (At the end of his arm, you can choose among several ends: a lamp, a hand gripper, ...) thanks to OLLO rivets that are very simple to remove. These rivets can be removed and added very quickly with the OLLO tool, which allows great design freedom. The software used for other creatures Poppy are the same as for the Poppy ergo, and also has a simulator.

BOM for Poppy Ergo Jr 1.0

For building a Poppy Ergo Jr, you will need:

- 6x robotis XL-320
- 6x XL cable (if possible 1 longer for the base)
- 1x USB2Dynamixel
- 1x TTL cable
- 1 external power supply able to output 7.5V at least 2A for the motors (see here for details)
- 1x small electronic board (see the electronic section)
- the 3D printed parts: [STL] [BOM]
- many OLLO rivets (between 4 and 12 per motor)
- 4x OLLO Pulley-L and their rivets
- 1x OLLO Tool
- 1x Raspberry Pi 2 with its power supply

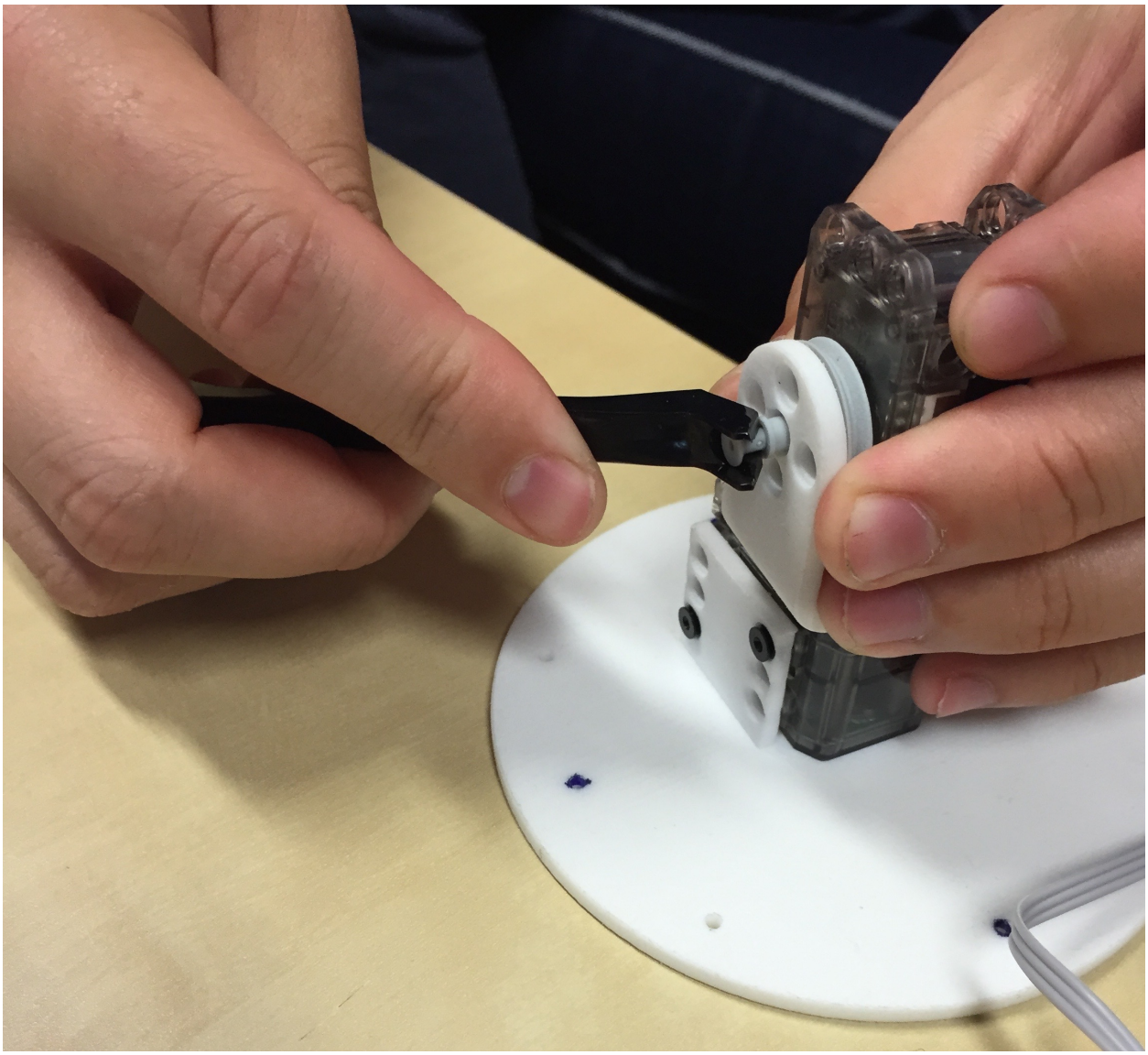
All the links to products we provide are from our local distributor but feel free to choose your own.

Mechanical Assembly

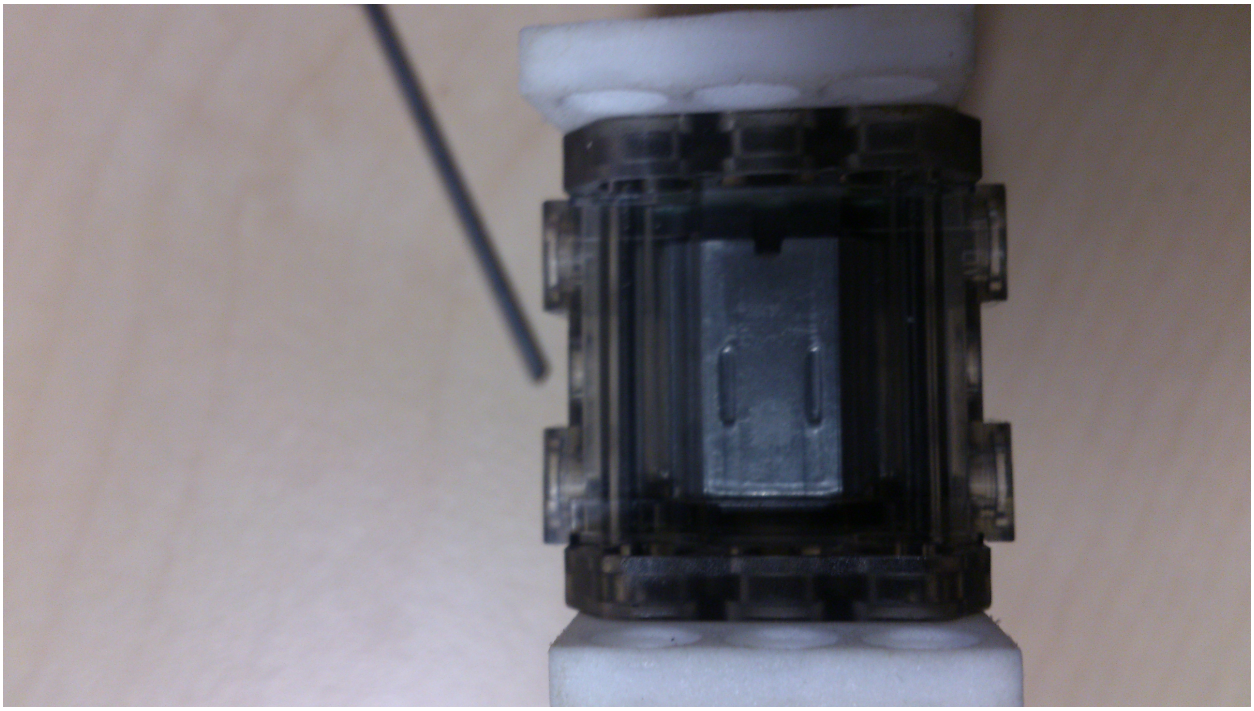
General advices

- You can assemble all the rivets before the construction, it will be easier after.
- You have to put the edges of the first part in the second part holes. This will then be able to remove them.

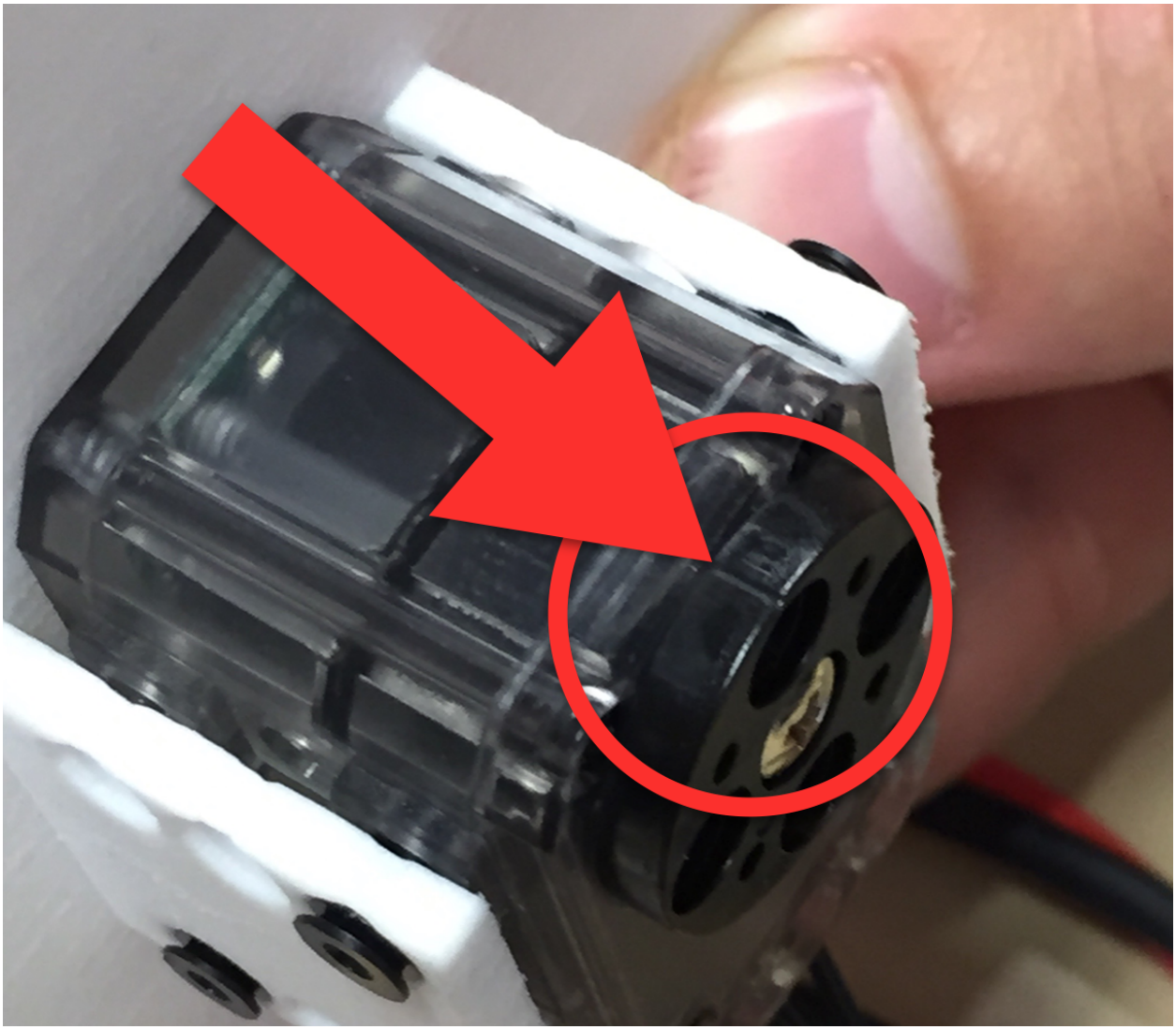
After, use the OLLO Tool for the rivets, it's really convenient. This tool allows to put and remove the rivets easily.



- Do not forget to put wires between motors while building the robot! Each motor, except the last, must have two wires; one connected to the previous motor and the other to the next (no favorite side).
- If you want, there are slots for getting the most beautifully wires and avoid damaging them.

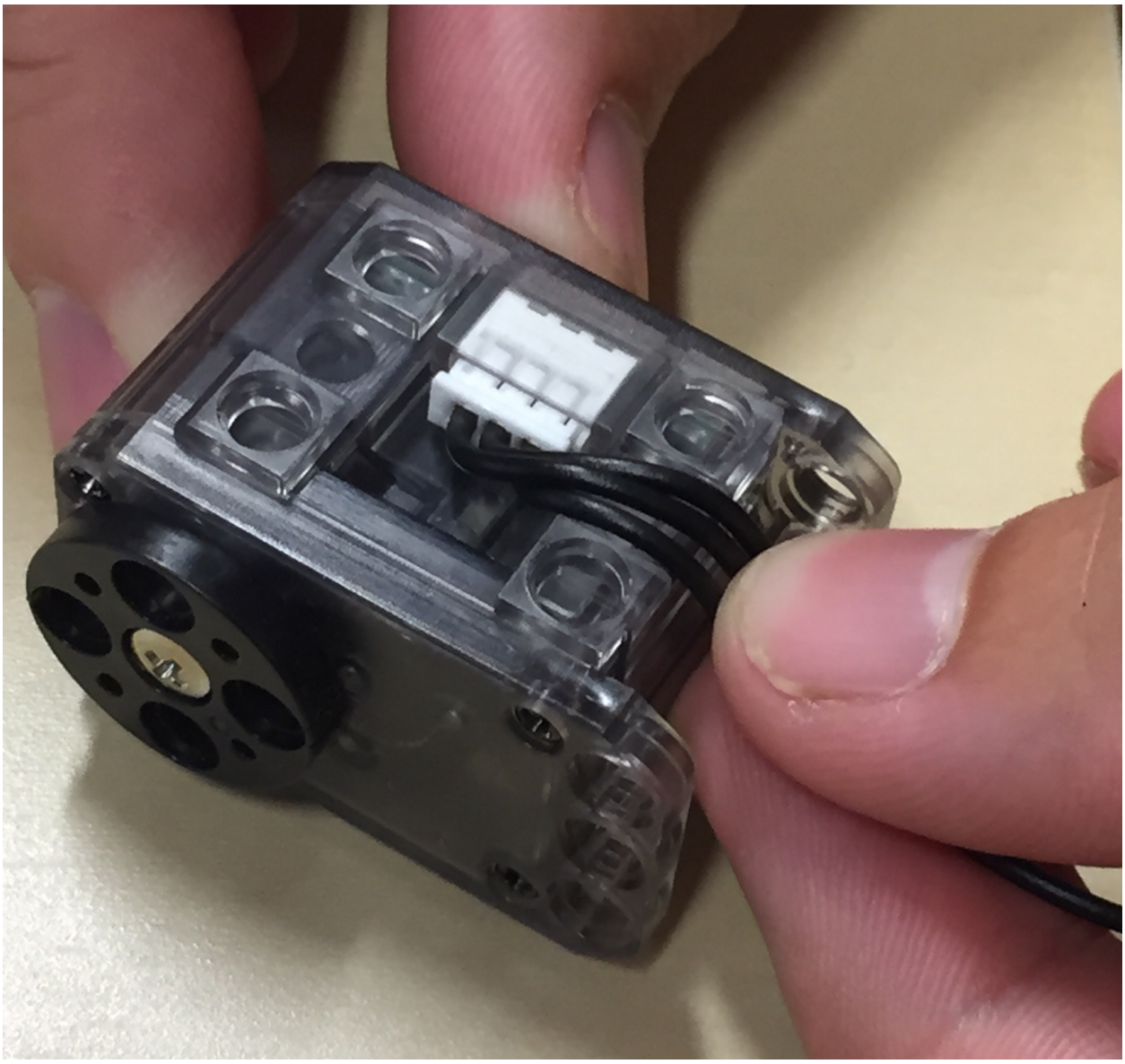


- Always align the horn with the motor before assembling them! Otherwise your Poppy Ergo Jr will look all weird.



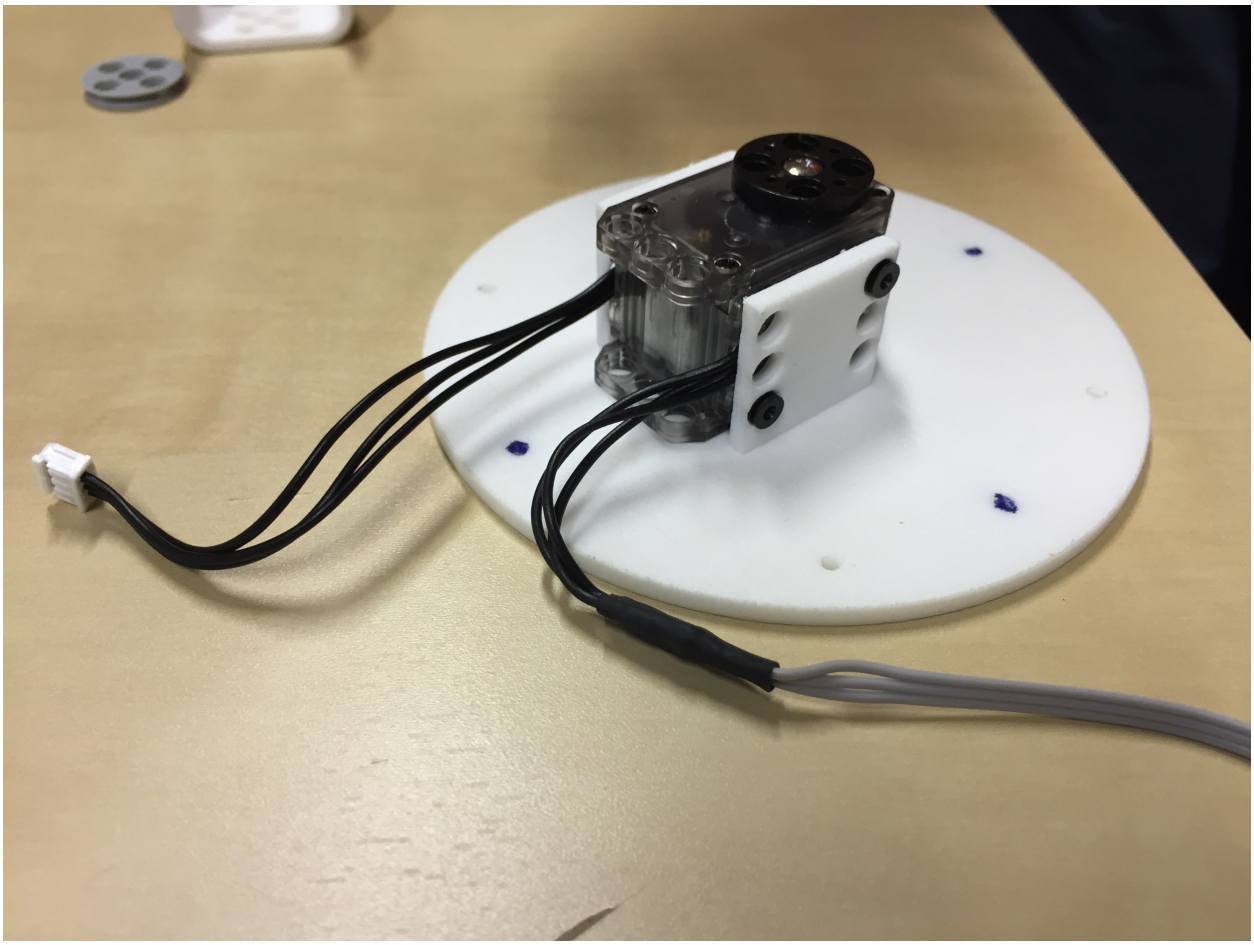
Step 1

Connect the longest cable to one XL-320 motor. Connect a regular cable to the other side.



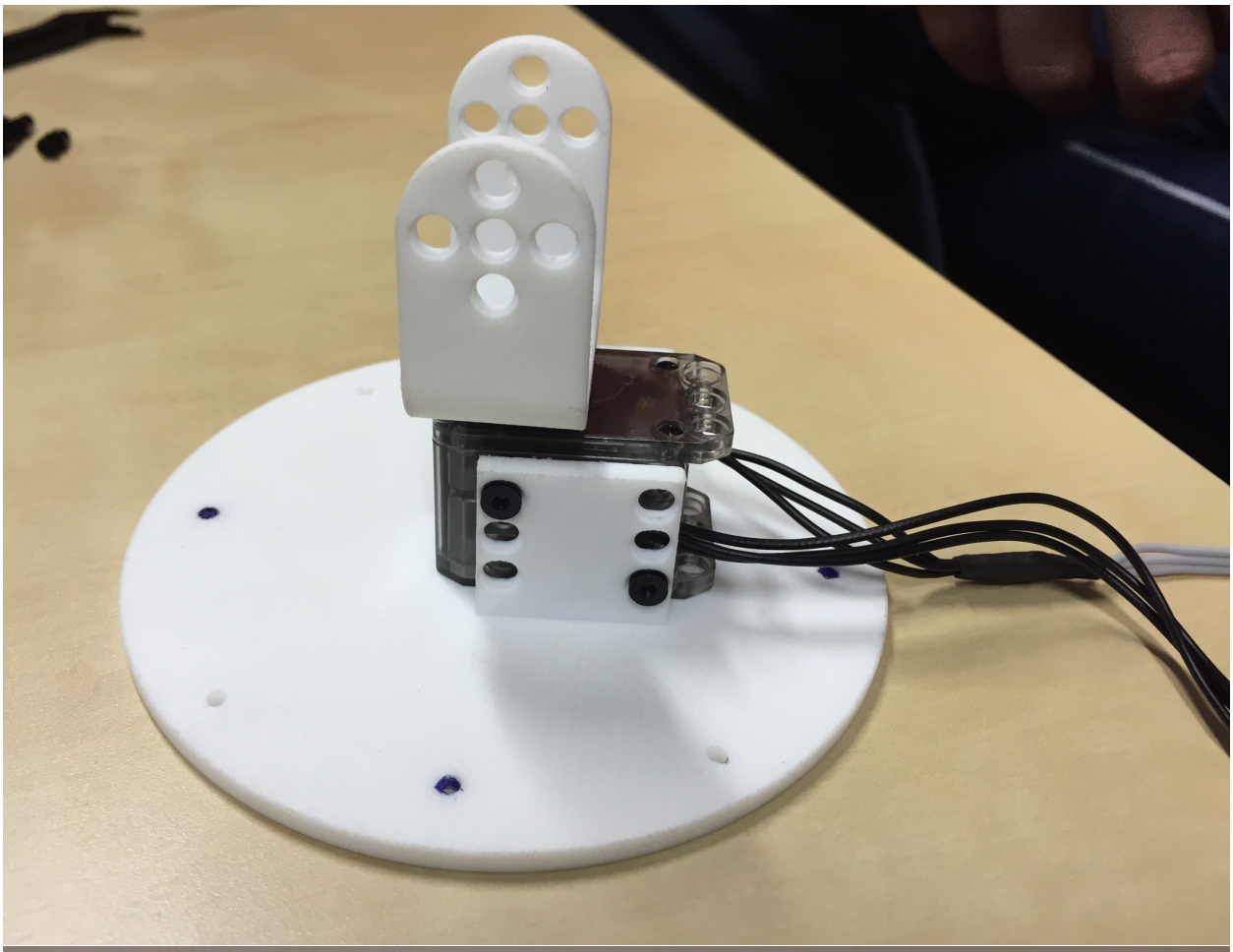
Step 2

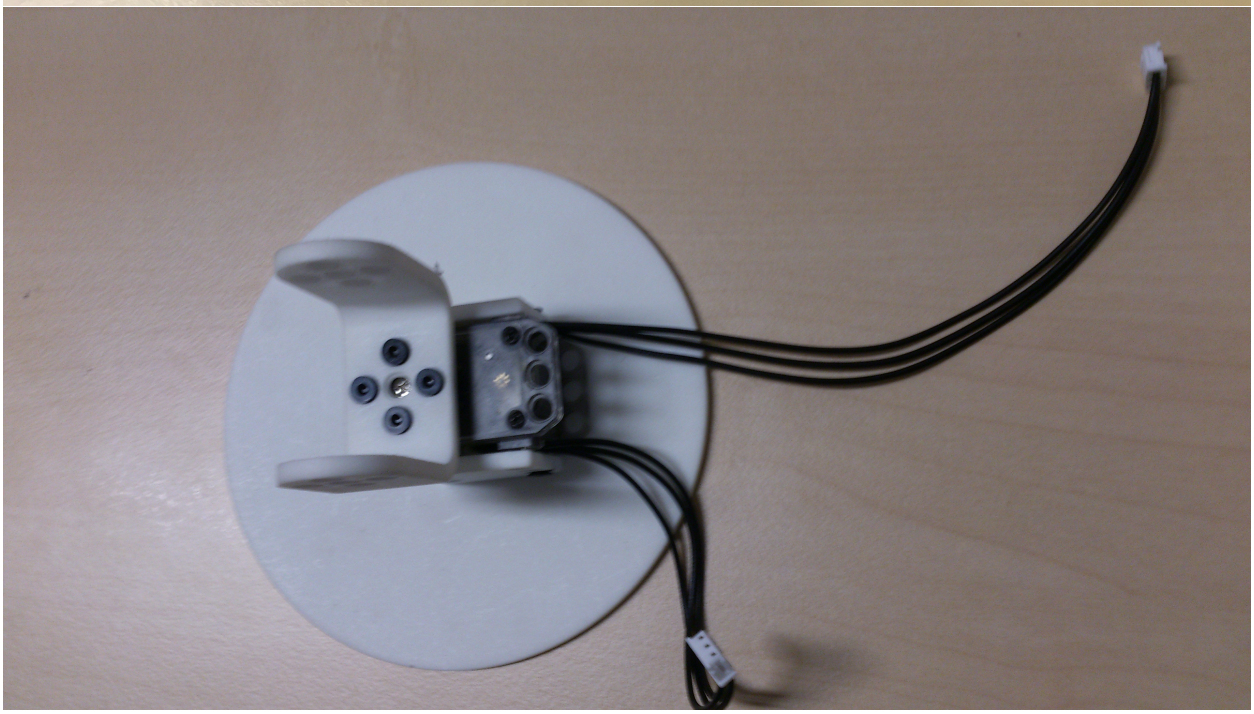
Mount the motor on the 3D printed base. The wires should go out from the back.



Step 3

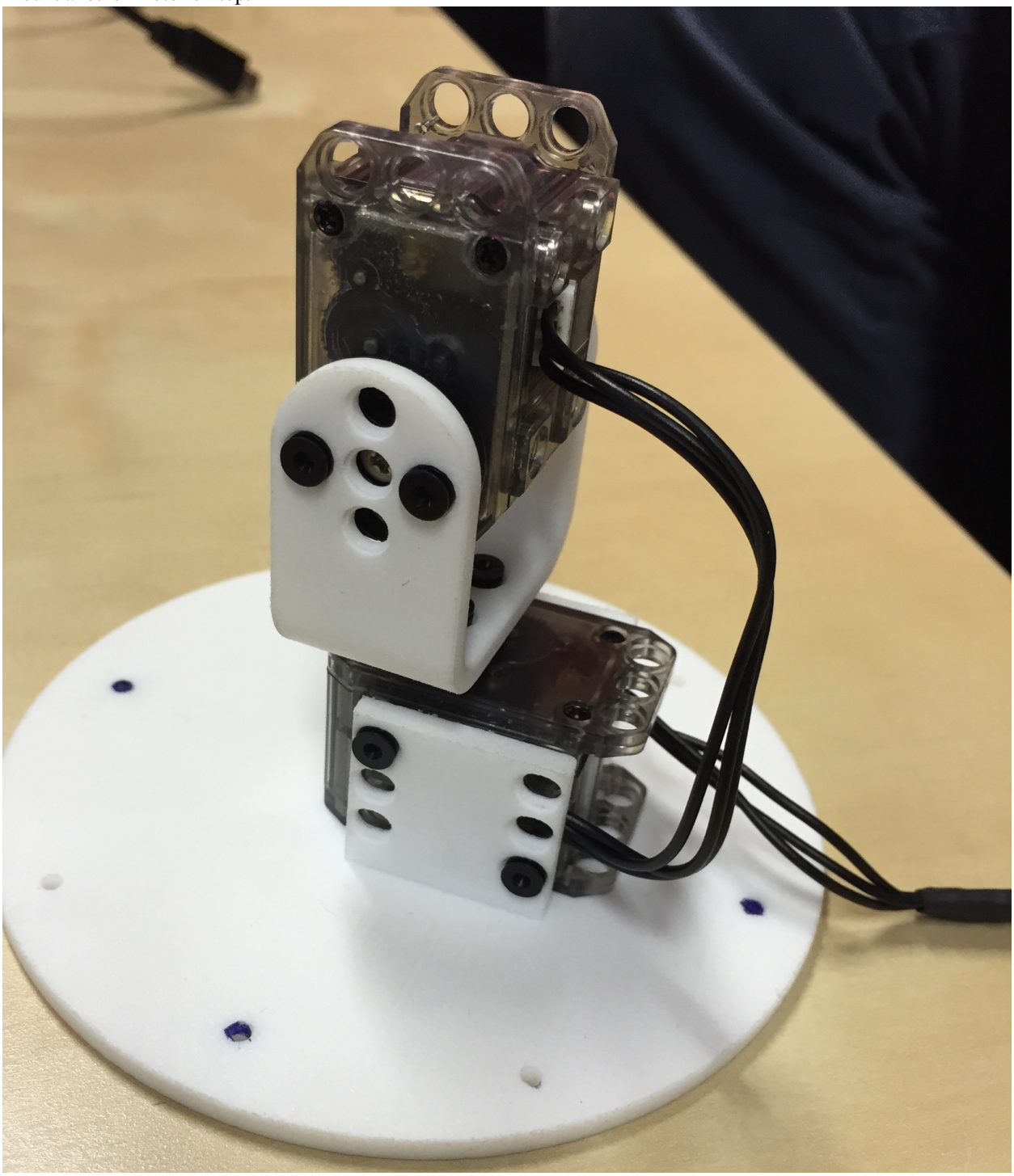
Mount the U_horn_to_horn part.



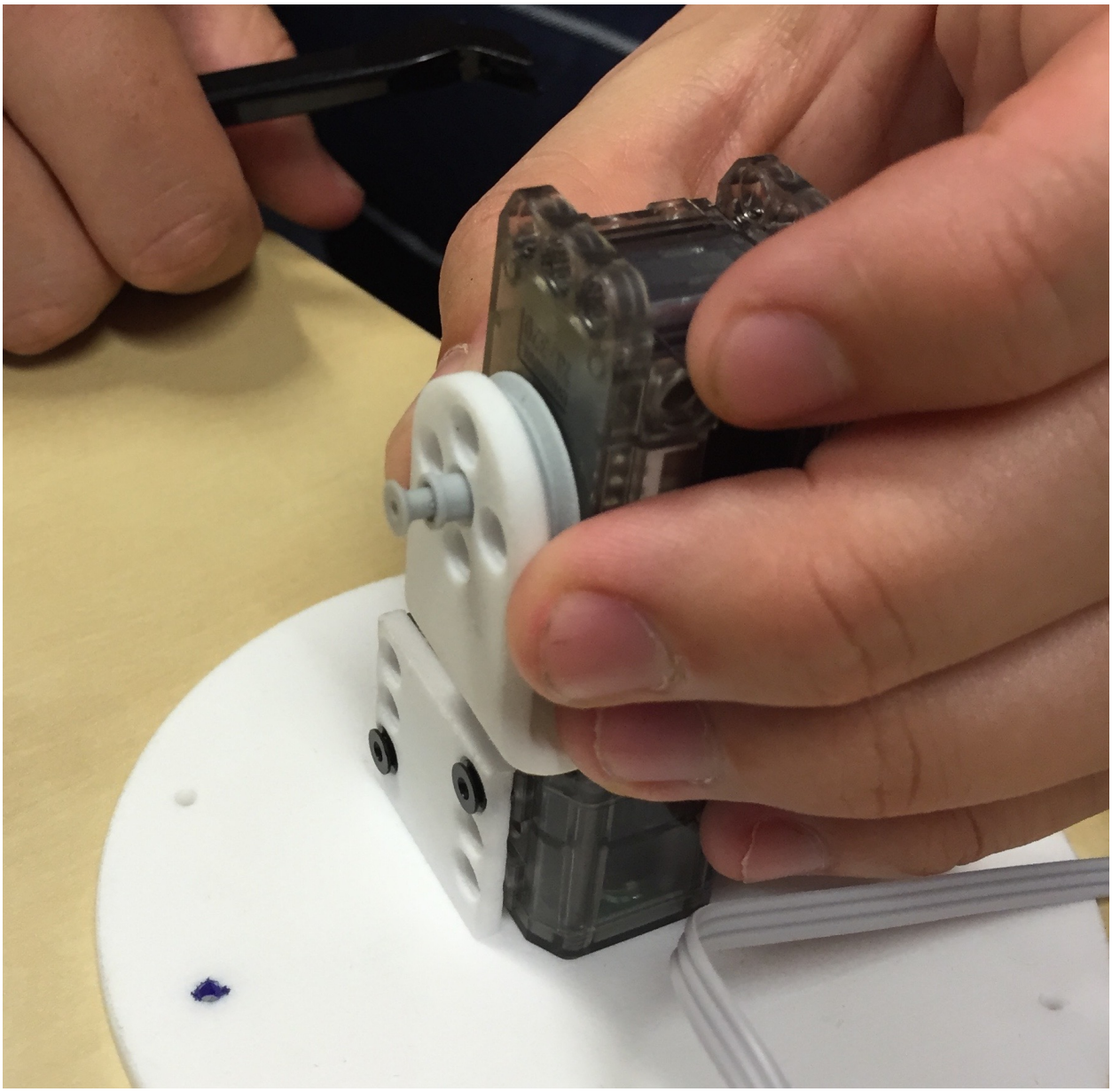


Step 4

Mount another motor on top.



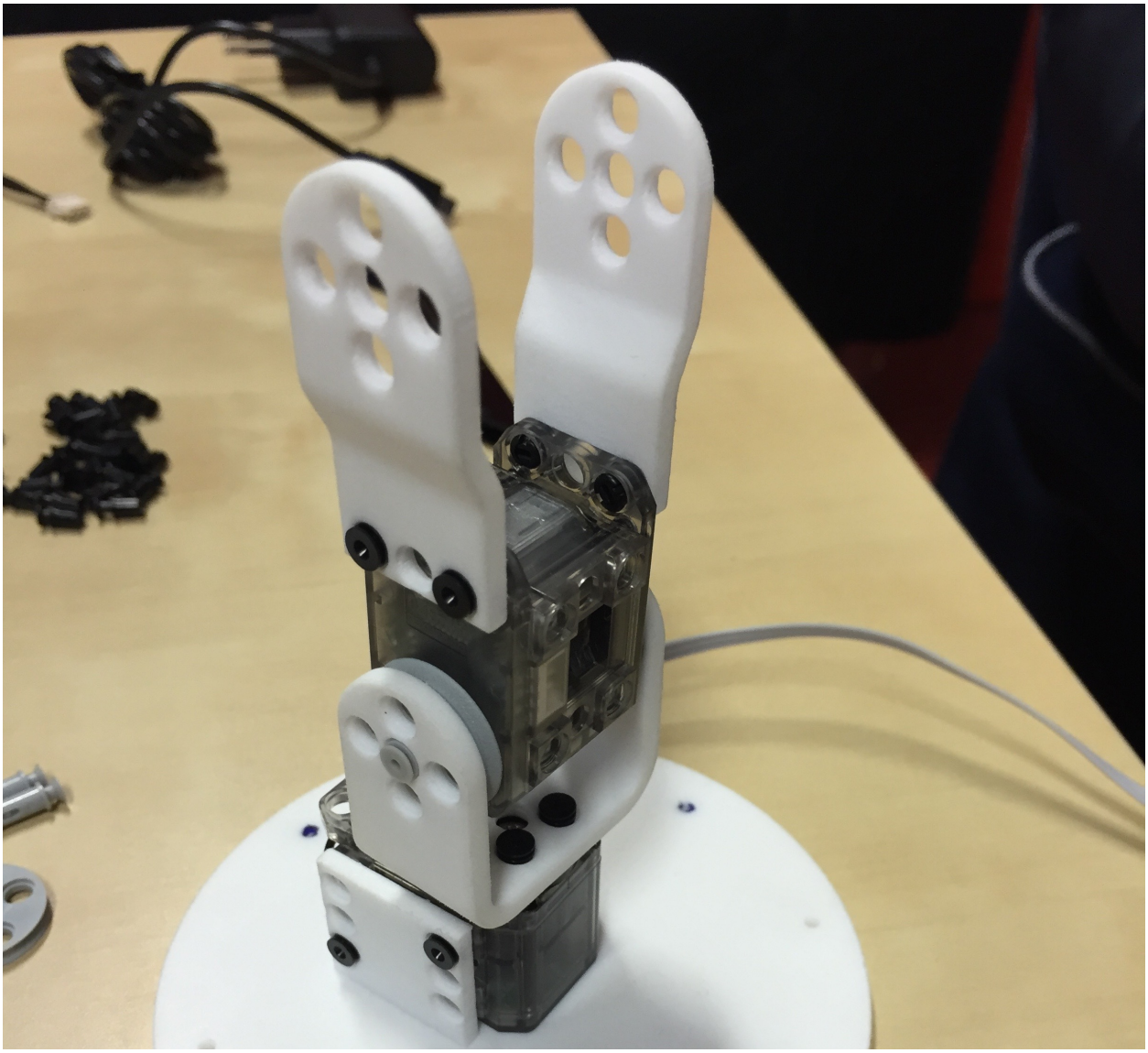
The OLLO Pulley is inside and you should assemble it with the bigger rivet. Bonus, you can watch this step in slow motion.



Do not forget the cable!

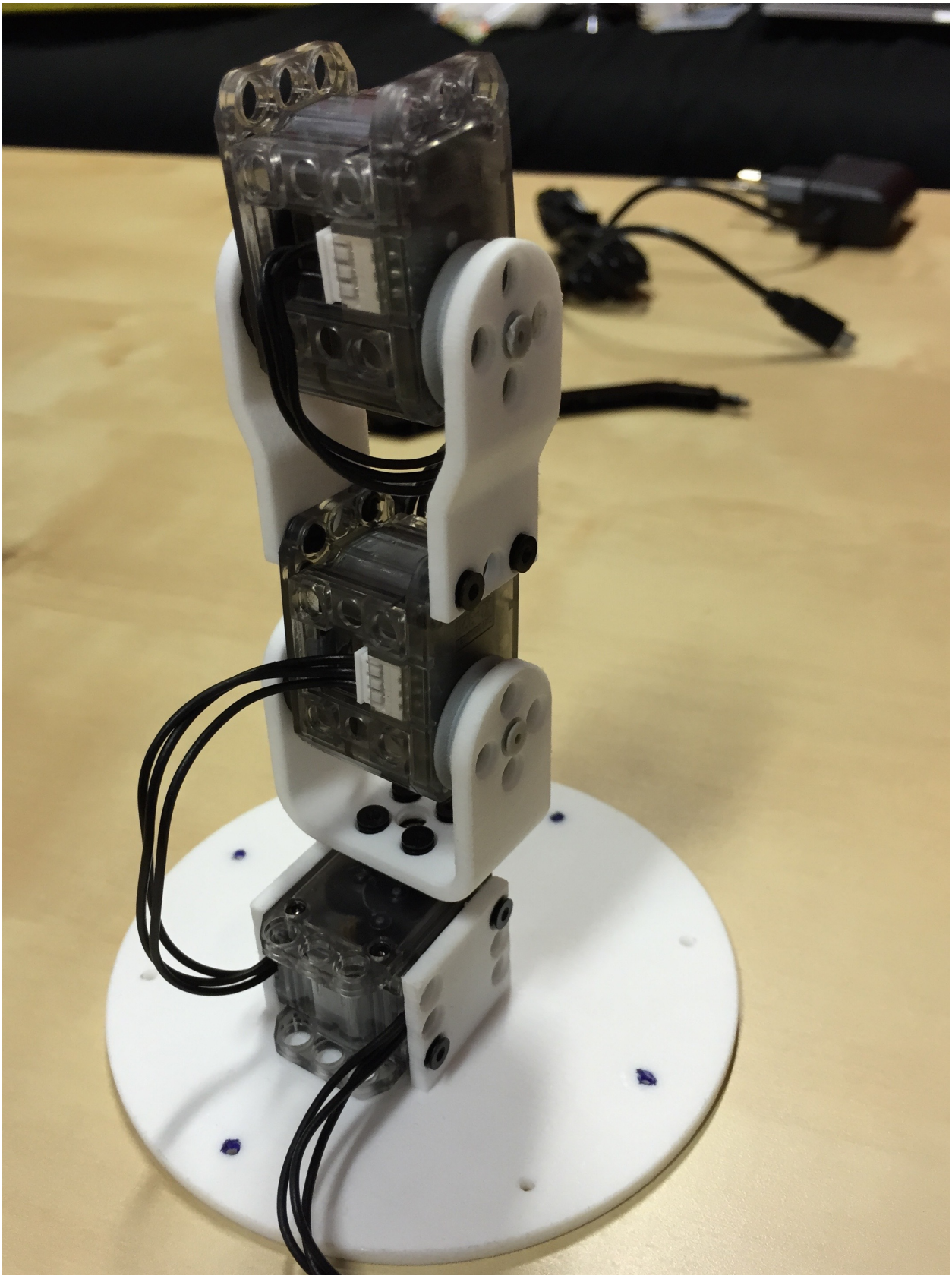
Step 5

Mount both shift_one_sides on the motors.



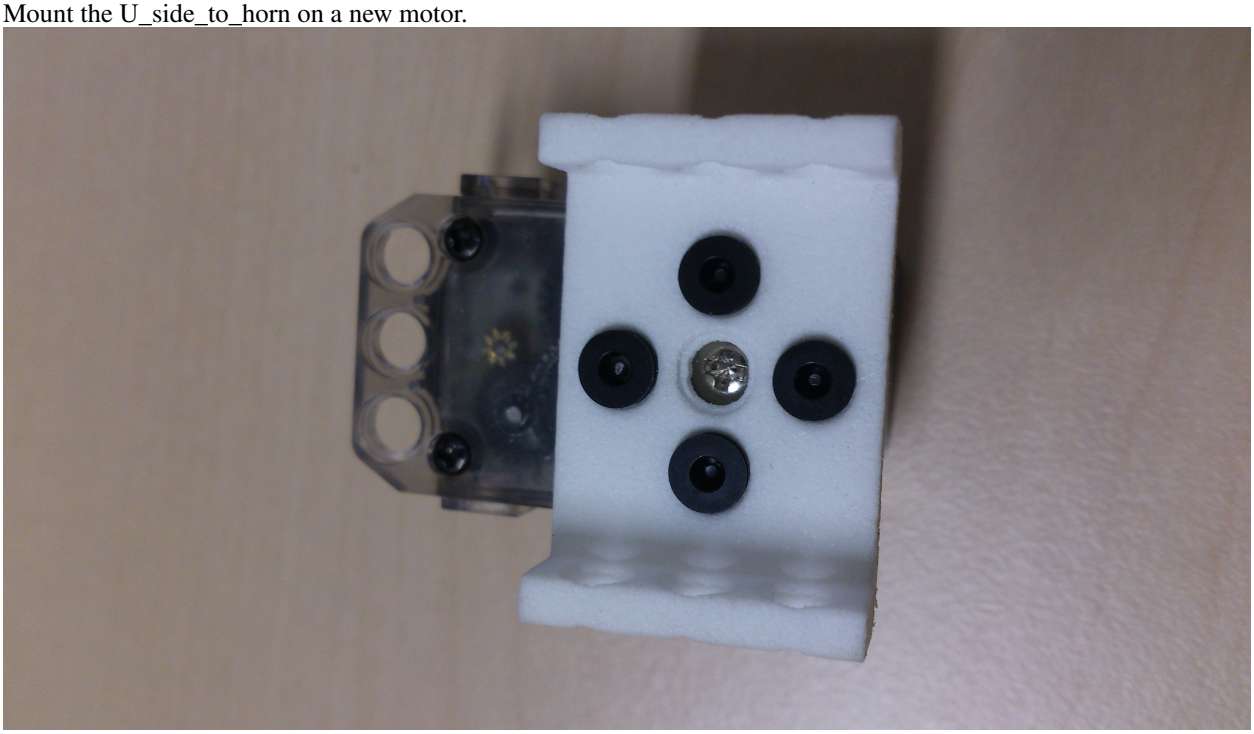
Step 6

Add another motor on top. Make sure to have the pulley on the same side.



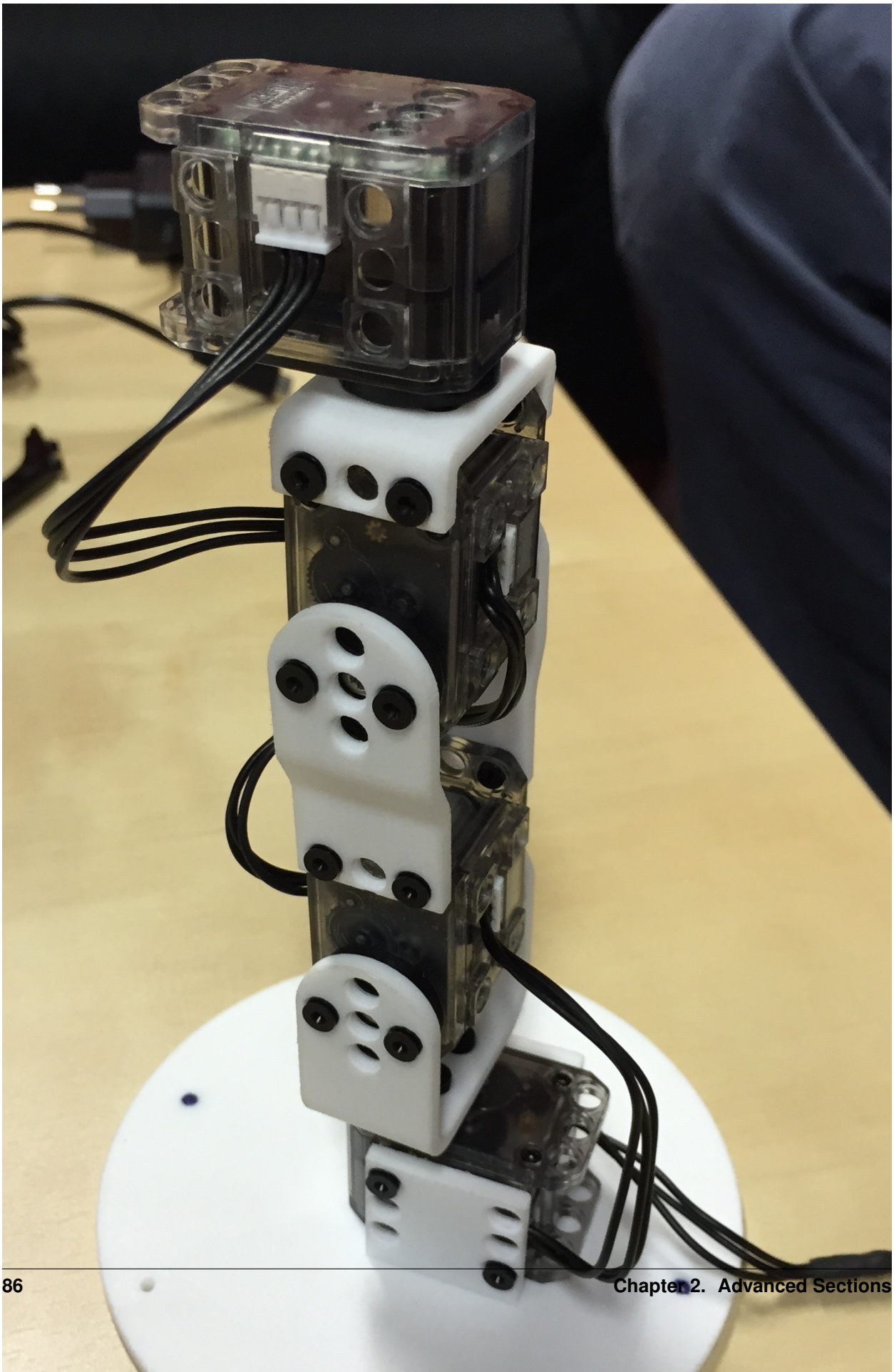
Step 7

Mount the U_side_to_horn on a new motor.



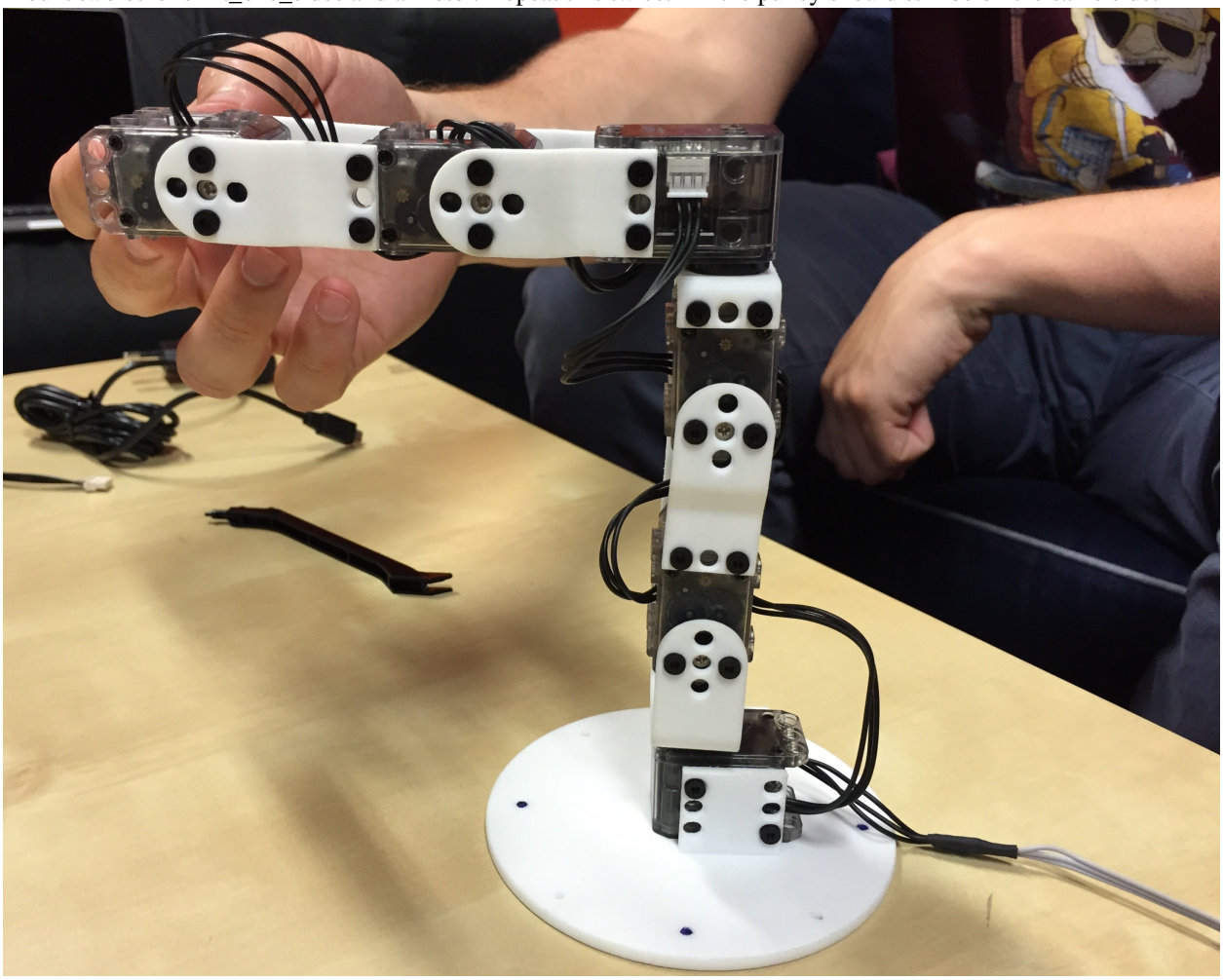
Step 8

Mount it on top of the previous assembly. The nose of the motor should be on the other side of the base.



Step 9

Mount two other shift_one_sides and a motor. Repeat this twice. All the pulley should still be on the same side.



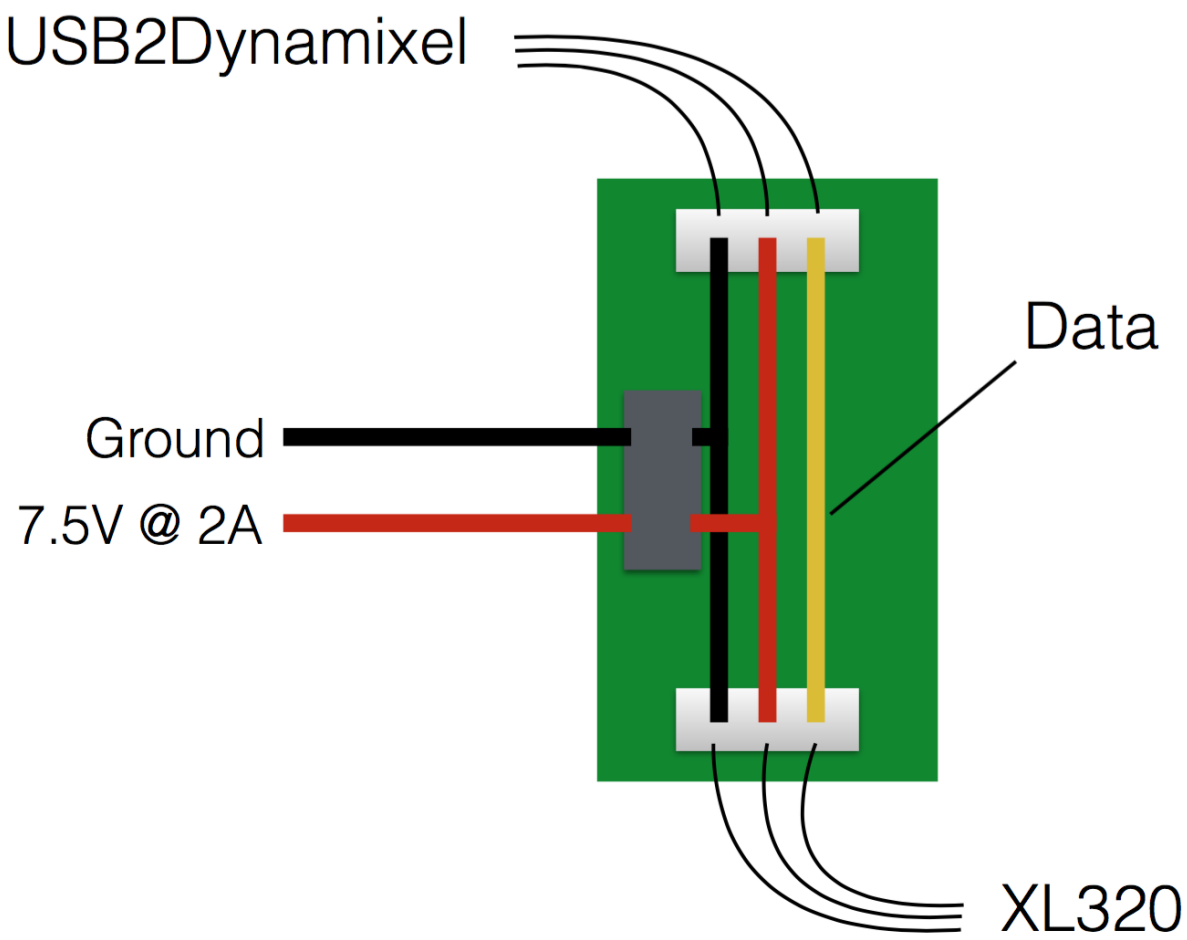
Step 10

Electronics A small bit of electronic hacking is required for now. You need to power the Xl320 motors with 7.5V.

The cables between motors have two purposes:

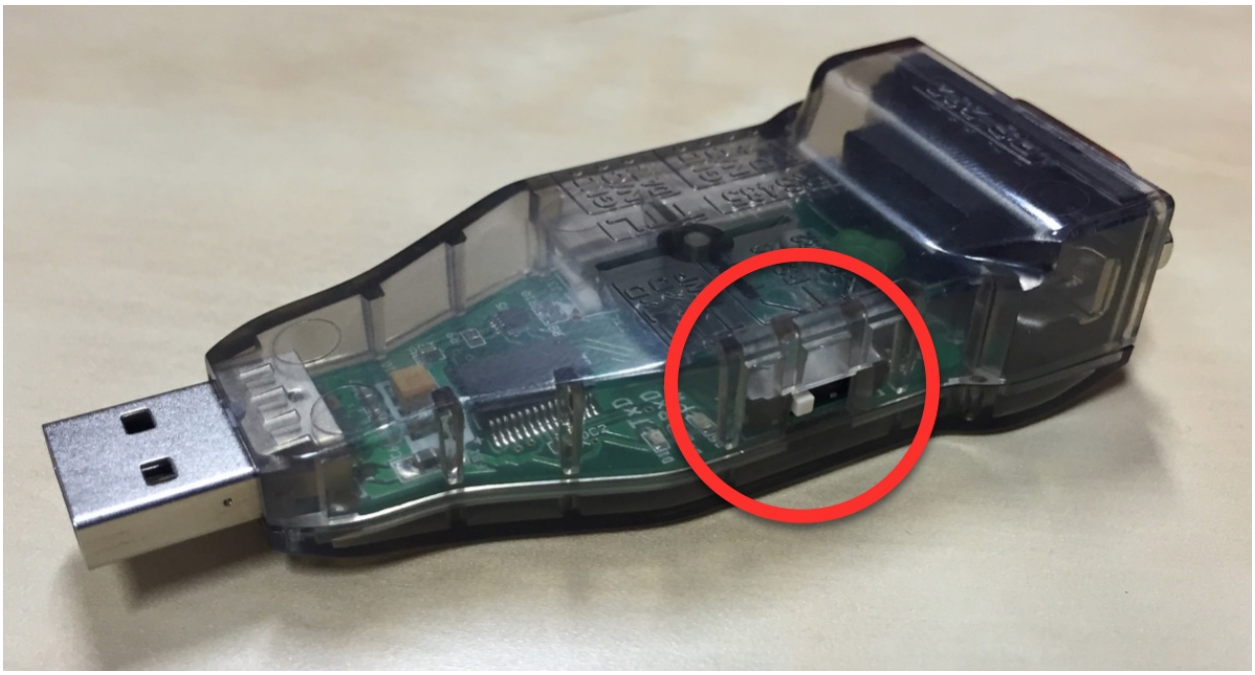
- distribute alimentation to each motor
- convey messages to each motor (ordering them to move or asking them for sensors' values)

Thus the alimentation should be added between the USB2Dynamixel (that deals with communication aspects) and the motors. To this end simply create the following hack.



Then connect the USB2Dynamixel on one end and the first motor on the other end. Connect the USB2Dynamixel to your computer or Raspberry Pi. And power the board with 7.5V.

The communication with the motor is TTL, thus configure the USB2Dynamixel in TTL mode as show below.



Step 11

Grab your favorite drink and relax.

2.2 Development Guides

2.2.1 Poppy-humanoid library

Introduction

Poppy Humanoid is the library allowing you to create a pypot.robot.robot.Robot corresponding to a standard Poppy Humanoid robot.

Using your Poppy Humanoid robot is as simple as:

```
from poppy_humanoid import PoppyHumanoid

poppy = PoppyHumanoid()

print poppy.motors
```

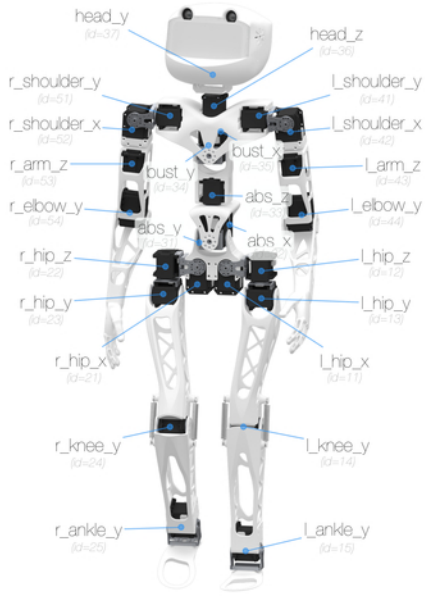
If you didn't change the default configuration file, this should connect to the 25 Dynamixel servomotors through two USB2AX.

Once the PoppyHumanoid object is created, you can start discovering it, control it at a basic level or using primitives

Poppy Humanoid robot overview

TODO

Robot's motors list



Poppy Humanoid specific features

poppy_humanoid.poppy_humanoid.PoppyHumanoid is not only a poppy.creatures.abstractcreature.AbstractPoppyCreature (which contains a pypot.robot.robot.Robot), it also have its own features and parameters.

The default goto behavior for each motor is set to 'minijerk' and each motor of the torso has the 'safe' compliant behavior (see the Pypot motor section), to avoid self-collision.

The following primitives are added to the robot (so you can directly use `poppy.sit_position.start()`):

- A poppy_humanoid.primitives.posture.StandPosition named `stand_position`: the robot moves to standing position (0 to all motors).
- A poppy_humanoid.primitives.posture.SitPosition named `sit_position`: the robot moves to sitting position (legs bent with heels under the buttock)
- A poppy_humanoid.primitives.safe.LimitTorquenamed `limit_torque`: Work In Progress. Each motor automatically sets the minimal torque needed to reach its goal position and therefore limit power consumption and motor heating. This primitive is looping.
- A poppy_humanoid.primitives.safe.TemperatureMonitor named `temperature_monitoring`: check the temperature of all motors and plays a sound if one is too hot. This primitive is looping and started by default.
- A poppy_humanoid.primitives.dance.SimpleBodyBeatMotion named `dance_beat_motion`: Simple primitive to make Poppy shake its upper body following a given beat rate in bpm. This primitive is looping.
- A poppy_humanoid.primitives.idle.UpperBodyIdleMotion named `upper_body_idle_motion`: Slow and small movements of the upper body to have the robot look less 'dead'. This primitive is looping.
- A poppy_humanoid.primitives.idle.HeadIdleMotion named `dance_beat_motion`: Slow and small movements of the head to have the robot look less 'dead'. This primitive is looping.
- A poppy_humanoid.primitives.interaction.ArmsTurnCompliant named `arms_turn_compliant`: Automatically turns the arms compliant when a force is applied. This primitive is looping.
- A poppy_humanoid.primitives.interaction.PuppetMaster named `arms_copy_motion`: Apply the motion made on the left arm to the right arm. This primitive is looping.

Remember to remove compliance before starting the primitives!

Installing

To install the Poppy Humanoid library, you can use pip:

```
pip install poppy-humanoid
```

Then you can update it with:

```
pip install --upgrade poppy-humanoid
```

If you prefer to work from the sources (latest but possibly unstable releases), you can clone them from Github and install them with (in the software folder):

```
python setup.py install
```

The requirements for Poppy Humanoid are Pypot and Poppy Creatures.

Poppy Humanoid motors List

Head

| Sub-assembly name | Motor name | Type | ID | | ———————— | ———- | —— | — | | Head | head_y | AX-12A | 37 | | Chest | head_z | AX-12A | 36 |

Trunk

| Sub-assembly name | Motor name | Type | ID | ### # | Double MX64 | abs_y | MX-64AT | 31 | ### ## | Double MX64 | abs_x | MX-64AT | 32 | ### ## | Spine | abs_z | MX-28AT | 33 | ### | Double MX28 | bust_y | MX-28AT | 34 | ### ## | Double MX28 | bust_x | MX-28AT | 35 | ### ## | Chest | l_shoulder_y | MX-28AT | 41 | ### ## | Chest | r_shoulder_y | MX-28AT | 51 |

Arms

| Sub-assembly name | Motor name | Type | ID | ### # | Left upper arm/shoulder | l_shoulder_x | MX-28AT | 42 | ### ## | Left upper arm | l_arm_z | MX-28AT | 43 | ### ## | Left upper arm | l_elbow_y | MX-28AT | 44 | ### ## | Right upper arm/shoulder | r_shoulder_x | MX-28AT | 52 | ### ## | Right upper arm | r_arm_z | MX-28AT | 53 | ### ## | Right upper arm | r_elbow_y | MX-28AT | 54 | ### ##

Legs

| Sub-assembly name | Motor name | Type | ID | ### # | Pelvis | l_hip_x | MX-28AT | 11 | ### ## | Pelvis | r_hip_x | MX-28AT | 21 | ### ## | Left hip | l_hip_z | MX-28AT | 12 | | Left hip | l_hip_y | MX-64AT | 13 | ### ## | Left thigh | l_knee_y | MX-28AT | 14 | ### ## | Left shin | l_ankle_y | MX-28AT | 15 | ### ## | Right hip | r_hip_z | MX-28AT | 22 | ### ## | Right hip | r_hip_y | MX-64AT | 23 | | Right thigh | r_knee_y | MX-28AT | 24 | ### ## | Right shin | r_ankle_y | MX-28AT | 25 | ### ##

2.2.2 Poppy-torso library

Introduction

Poppy Torso is the library allowing you to create a pypot.robot.robot.Robot corresponding to a standard Poppy Torso robot.

Using your Poppy Torso robot is as simple as:

```
from poppy_torso import PoppyTorso

poppy = PoppyTorso()

print poppy.motors
```

If you didn't change the default configuration file, this should connect to the 13 Dynamixel servomotors through the USB2AX.

Once the PoppyTorso object is created, you can start discovering it, control it at a basic level or using primitives

The sources are available on Github.

Poppy Torso robot overview

TODO

image motors

list motors, move, etc

Poppy Torso specific features

poppy_torso.poppy_torso.PoppyTorso is not only a poppy.creatures.abstractcreature.AbstractPoppyCreature (which contains a pypot.robot.robot.Robot), it also have its own features and parameters.

The default goto behavior for each motor is set to 'minijerk'.

The following primitives are added to the robot (so you can directly use `poppy.limit_torque.start()`):

- A poppy_torso.primitives.safe.LimitTorque named `limit_torque`: Work In Progress. Each motor automatically sets the minimal torque needed to reach its goal position and therefore limit power consumption and motor heating. This primitive is looping.
- A poppy_torso.primitives.safe.TemperatureMonitor named `temperature_monitoring`: check the temperature of all motors and plays a sound if one is too hot. This primitive is looping and started by default.
- A poppy_torso.primitives.dance.SimpleBodyBeatMotion named `dance_beat_motion`: Simple primitive to make Poppy shake its upper body following a given beat rate in bpm. This primitive is looping.
- A poppy_torso.primitives.idle.UpperBodyIdleMotion named `upper_body_idle_motion`: Slow and small movements of the upper body to have the robot look less 'dead'. This primitive is looping.
- A poppy_torso.primitives.idle.HeadIdleMotion named `dance_beat_motion`: Slow and small movements of the head to have the robot look less 'dead'. This primitive is looping.
- A poppy_torso.primitives.interaction.ArmsTurnCompliant named `arms_turn_compliant`: Automatically turns the arms compliant when a force is applied. This primitive is looping.
- A poppy_torso.primitives.interaction.PuppetMaster named `arms_copy_motion`: Apply the motion made on the left arm to the right arm. This primitive is looping.

Remember to remove compliance before starting the primitives!

Installing

To install the Poppy Torso library, you can use pip:

```
pip install poppy-torso
```

Then you can update it with:

```
pip install --upgrade poppy-torso
```

If you prefer to work from the sources (latest but possibly unstable releases), you can clone them from Github and install them with (in the software folder):

```
python setup.py install
```

The requirements for Poppy Torso are Pypot and Poppy Creatures.

2.2.3 Poppy-ergo-jr library

Introduction

Poppy Ergo Jr is the library allowing you to create a pypot.robot.robot.Robot corresponding to a standard Poppy Ergo Jr robot.

Using your Poppy Torso robot is as simple as:

```
from poppy_ergo_jr import PoppyErgoJr

poppy = PoppyErgoJr()

print poppy.motors
```

If you didn't change the default configuration file, this should connect to the 6 Dynamixel servomotors through the USB2AX.

Once the PoppyErgoJr object is created, you can start discovering it, control it at a basic level or using primitives

Poppy Ergo Jr robot overview

TODO

image motors

list motors, move, etc

Poppy Ergo Jr specific features

poppy_ergo_jr.poppy_ergo_jr.PoppyErgoJr is not only a poppy.creatures.abstractcreature.AbstractPoppyCreature (which contains a pypot.robot.robot.Robot), it also have its own features and parameters.

The motors with IDs 0, 2 and 4 have their max_torque set to 0.7, so they can use only 70% of their maximal power.

The following primitives are added to the robot (so you can directly use `poppy.base_posture.start()`):

- A poppy_ergo_jr.primitives.postures.BasePosture named `base_posture`: the robot goes in an initial posture.
- A poppy_ergo_jr.primitives.postures.RestPosture named `rest_posture`: the robot goes in a 'resting' posture.
- A poppy_ergo_jr.primitives.dance.Dance named `dance`: Simple dance based on sinusoidal motions. This primitive is looping.
- A poppy_ergo_jr.primitives.jump.Jump named `jump`: The robot folds on itself, then moves quickly, which results in a jump (if the robot base if not too heavy).

If the robot is simulated, another primitive is launched:

- A poppy_ergo_jr.primitives.headfollow.HeadFollow named `head_follow`: The robot's *head*, or end-point, follows a marker from the simulator. This primitive is looping.

Remember to remove compliance before starting the primitives!

Installing

To install the Poppy Ergo Jr library, you can use pip:

```
pip install poppy-ergo-jr
```

Then you can update it with:

```
pip install --upgrade poppy-ergo-jr
```

If you prefer to work from the sources (latest but possibly unstable releases), you can clone them from Github and install them with (in the software folder):

```
python setup.py install
```

The requirements for Poppy Ergo Jr are Pypot and Poppy Creatures.

2.2.4 Poppy-creature library

Introduction

Poppy-creature is a small library providing a link between specific robots (Poppy Humanoid, Poppy Ergo JR...) and Pypot, the generic, lower level library.

It mainly contains the class definition of poppy.creatures.abstractcreature.AbstractPoppyCreature which takes a configuration and builds a pypot.robot.robot.Robot out of it, but also a bunch of parameters to launch Snap! or HTTP servers, or to replace the communication toward Dynamixel servos by a communication with a simulator.

The arguments you can provide are:

- `base_path` default: None Path where the creature sources are. The librarie looks in the default PATH if not set.
- `config` default: None Path to the configuration file with respect to the base-path
- `simulator` default: None Possible values : 'vrep' or 'threejs'. Defines if we are using a simulator (and which one) or a real robot.
- `scene` default: None Path to the scene to load in the simulator. Only if simulator is vrep. Defaults to the scene present in the creature library if any (e.g. poppy_humanoid.ttt).
- `host` default: 'localhost' Hostname of the machine where the simulator runs. Only if simulator is not None.
- `port` default: 19997 Port of the simulator. Only if simulator is not None.
- `use_snap` default: False Should we launch the Snap! server
- `snap_host` default: 0.0.0.0 Hostname of the Snap! server
- `snap_port` default: 6969 Port of the Snap! server
- `snap_quiet` default: True Should Snap! not output logs
- `use_http` default: False Should we launch the HTTP server (for REST API use)
- `http_host` default: 0.0.0.0 Hostname of the HTTP server
- `http_port` default: 8080 Port of the HTTP server

- `http_quiet` default: True Should HTTP not output logs
- `use_remote` default: False Should we launch the Remote Robot server (for REST API use)
- `remote_host` default: 0.0.0.0 Hostname of the Remote Robot server
- `remote_port` default: 4242 Port of the Remote Robot server
- `sync` default: True Should we launch the synchronization loop for motor communication (see the Dynamixel low-level Pypot section)

The sources are available on Github.

Poppy services

But poppy-creature also provides a set of very useful services that can be launched directly from the command line inside your robot if you installed the soft from poppy_install . Example:

```
poppy-services poppy-humanoid --snap --no-browser
```

This will launch the snap server for a Poppy Humanoid robot without opening the browser page for Snap! (if you have a screen, mouse and keyboard connected directly on the head of the robot, you can remove this argument, but in most cases, you launch this inside the robot through SSH and then connect in a browser from another computer).

The way to use it is:

```
poppy-services <creature_name> <options>
```

the available options are:

- `--vrep`: creates the specified creature for using with V-rep simulator
- `--threejs`: creates the specified creature for using with Three.js simulator (in-browser 3D modelisation) and also maunches the HTTP server needed by the Three.js simulation.
- `--snap`: launches the Snap! server and directly imports the specific Poppy blocks.
- `-nb` or `--no-browser`: avoid automatic start of Snap! in web browser, use only with –snap
- `--http`: start a http robot server
- `--remote`: start a remote robot server
- `-v` or `--verbose`: start services in verbose mode (more logs)

Create your own Poppy creature

While developping a new Poppy creature, it is first easier to simply define it in a configuration file or dictionnary and instanciate a pypot.robot.robot.Robot from Pypot directly (see the robot object from Pypot).

But when you want to make it easily usable and available to non-geek public, the best is to create your own creature's library. It should contain a configuration file and a class that extends ~poppy.creatures.abstractcreature.AbstractPoppyCreature. You can then add your own properties and primitives.

Example from Poppy Humanoid:

```
class PoppyHumanoid(AbstractPoppyCreature):
    @classmethod
    def setup(cls, robot):
        robot._primitive_manager._filter = partial(numpy.sum, axis=0)

        for m in robot.motors:
```

```
        m.goto_behavior = `minjerk'

    for m in robot.torso:
        m.compliant_behavior = `safe'

    # Attach default primitives:
    # basic primitives:
    robot.attach_primitive(StandPosition(robot), `stand_position')
    robot.attach_primitive(SitPosition(robot), `sit_position')

    # Safe primitives:
    robot.attach_primitive(LimitTorque(robot), `limit_torque')
```

Package your code it properly using setuptools . For a better integration with the Poppy installer scripts, please have in the root of your repo a folder named *software* containing:

- the installation files (setup.py, MANIFEST, LICENCE)
- a folder named poppy_yourcreaturename containing your actual code

At the end, don't forget to share it to the community! Most interesting creatures will be added to this documentation!

Installing

To install the Poppy Creature library, you can use pip:

```
pip install poppy-creature
```

Then you can update it with:

```
pip install --upgrade poppy-creature
```

If you prefer to work from the sources (latest but possibly unstable releases), you can clone them from Github and install them with (in the software folder):

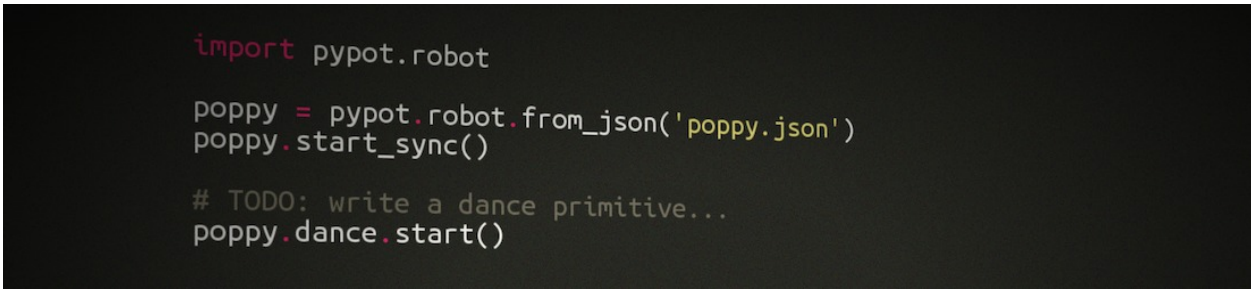
```
python setup.py install
```

The requirements for Poppy Creature are Pypot and bottle.

2.2.5 Pypot library

Introduction

What is pypot?



Pypot is a framework developed in the Inria FLOWERS team to make it easy and fast to control custom robots based on dynamixel motors. This framework provides different level of abstraction corresponding to different types of use. More precisely, you can use pypot to:

- directly control robotis motors through a USB2serial device (both protocols v1 and v2 are supported: you can use it with AX, RX, MX and XL320 motors),
- define the structure of your particular robot and control it through high-level commands.
- define primitives and easily combine them to create complex behavior.
- [work in progress] define sensors and integrate them to the creature's control.

Pypot has been entirely written in Python to allow for fast development, easy deployment and quick scripting by non-necessary expert developers. The serial communication is handled through the standard library and thus allows for rather high performance (10ms sensorimotor loop).

It is crossed-platform and has been tested on Linux, Windows and Mac OS. It is distributed under the GPL V3 open source license.

Pypot is also compatible with the V-REP simulator and allows you to seamlessly switch from a real robot to its simulated equivalent without having to modify your code.

note

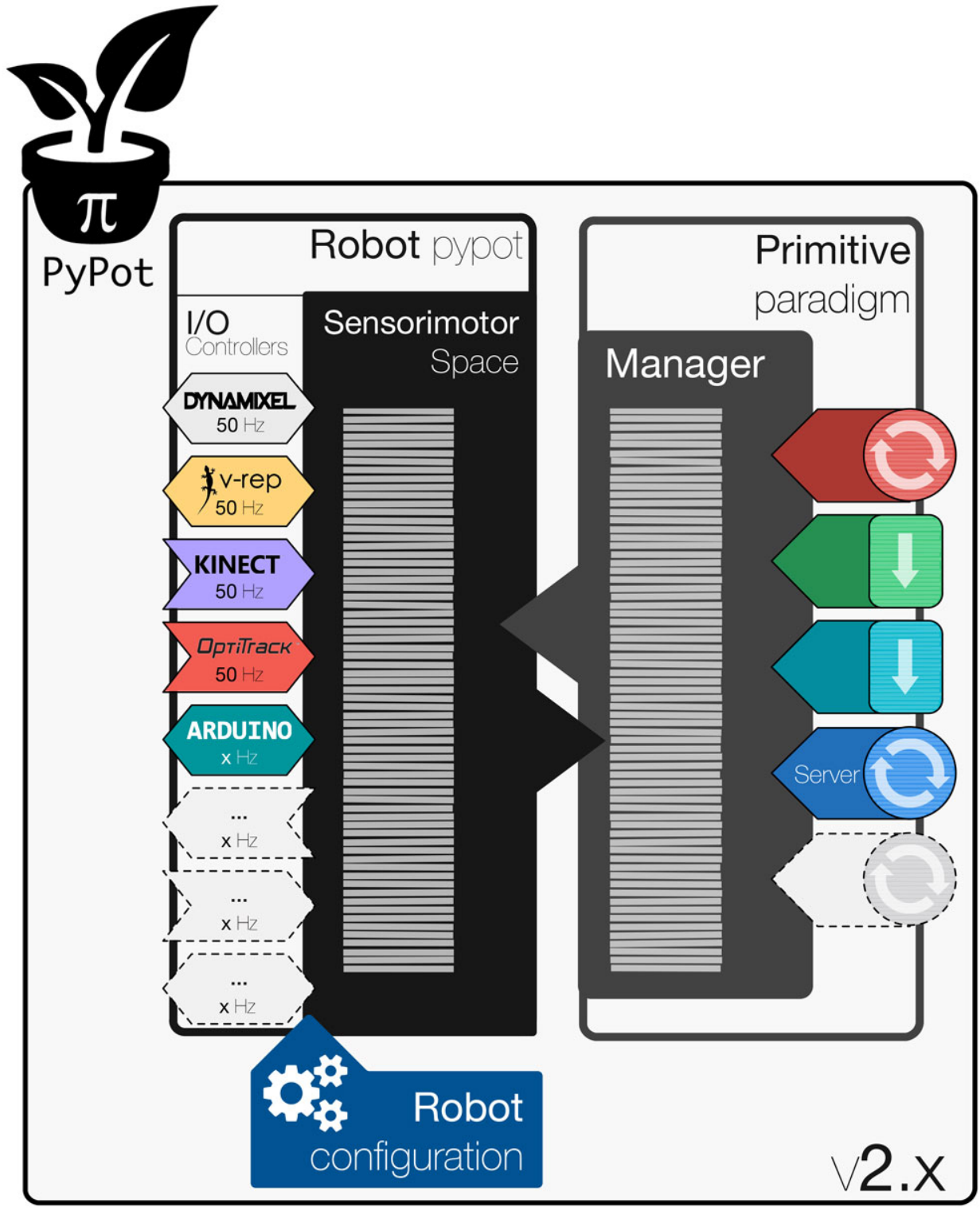
The other libraries from the Poppy project (Poppy Humanoid : or Poppy Ergo Jr...) are built on top of pypot and abstract most of its operating and already come with convenient method for creating and starting your robot. So, when starting with a Poppy creature, we advise you to first discover the specific library and dive into Pypot only when you are ready to build advanced programs. Pypot is also a good starting point if you want to define your own Poppy Creatures.

Pypot's architecture

Pypot's architecture is built upon the following basic concepts:

- **I/O**: low-level layer handling the communication with motors or sensors. This abstract layer has been designed to be as generic as possible. The idea is to keep each specific communication protocol separated from the rest of the architecture and allow for an easy replacement of an IO by another one - such an example is detailed in the vrep section.
- **Motor** or **Sensor** : abstraction layer allowing to command the same type of devices in the same way. Each software motor or sensor is linked with its hardware equivalent.
- **Controller**: : set of update loops to keep an (or multiple) "hardware" device(s) up to date with their "software" equivalent, moreover when several devices (e.g. Dynamixel motors) share the same communication bus. This synchronization can goes only from the hard to the soft (e.g. in the case of a sensor) or both ways (e.g. for reading motor values and sending motor commands). The calls can be asynchronous or synchronous, each controller can have its own refresh frequency. An example of pypot.robot.controller is the pypot.dynamixel.controller.DxlController which synchronizes position/speed/load of all motors on a dynamixel bus in both directions. On the same bus, you can have several controllers of different frequencies.
- **Robot**: The robot layer is a pure abstraction which aims at bringing together different types of motors and sensors. This high-level is most likely to be the one accessed by the end-user which wants to directly control the motors of its robot no matter what is the IO used underneath. The robot can be directly created using a configuration file describing all IO and Controllers used.
- **Primitives**: independent behaviors applied to a robot. They are not directly accessing the robot registers but are first combined through a Primitive Manager which sends the results of this combination to the robot. This abstraction is used to designed behavioral-unit that can be combined into more complex behaviors (e.g. a walking primitive and and balance primitive combined to obtain a balanced-walking). Primitives are also a convenient way to monitor or remotely access a robot - ensuring some sort of sandboxing.

Those main aspects of pypot's architecture are summarized in the figure below.



Refer to this section to learn how to install pypot on your system.

Installation and updating

Installation with a pre-flashed system image If you are using an official Poppy creature (Poppy Humanoid, Poppy Torso, Poppy Ergo Jr), you may receive a ready-to-use, already-flashed SD card for your Raspberry Pi 2, with everything already installed.

If you are building your kit yourself, get a SD card (8 GB or more) and flash it with the TODO LINK system image, following for example these instructions

Now simply check your installation (TODO link network check) and, if needed, update it. Your Poppy creature is ready to come alive.

Manual installation

Why do this?

- Because no-one offers a system image for your Poppy creature
- Because you want to install the latest version of each library (even if they may be less stable)
- Because your creature's brain is not a Raspberry Pi. In fact, it may even be your desktop computer, where you directly pluggued the USB2serial device.
- Because you are using a simulator

Requirements Pypot is written in python and need a python interpreter to be run. Moreover pypot has scipy and numpy for dependencies, as they are not fully written in python they need system side packages to be build, it easier to use pre-build binaries for your operating system.

Windows The easier way is to install Anaconda a pre-packaged python distribution with lot of scientific librairies pre-compiled and a graphical installer.

After that, you can install pypot with *pip* in the command prompt.

GNU/Linux You can also install Anaconda, but it's faster to use the binaries provided by your default package manager.

On Ubuntu & Debian:

```
sudo apt-get install python-pip python-numpy python-scipy python-matplotlib
```

On Fedora:

```
sudo yum install python-pip numpy scipy python-matplotlib
```

On Arch Linux:

```
sudo pacman -S python2-pip python2-scipy python2-numpy python2-matplotlib
```

After that, you can install pypot with *pip*.

Mac OSX Mac OSX (unlike GNU/Linux distributions) don't come with a package manager, but there are a couple of popular package managers you can install, like Homebrew.

The easier way is to install Homebrew. You have to type these commands in a terminal:

```
ruby -e ``$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)'
```

An use Homebrew to install python:

```
brew install python
```

After that, you can install pypot with *pip*.

Via Python Packages The pypot package is entirely written in Python. So, the install process should be rather straightforward. You can directly install it via easy_install or pip:

```
pip install pypot
```

or:

```
easy_install pypot
```

The up to date archive can also be directly downloaded here.

If you are on a GNU/Linux operating system, you will need to execute the above commands with **sudo**.

From the source code You can also install it from the source. You can clone/fork our repo directly on github.

Before you start building pypot, you need to make sure that the following packages are already installed on your computer:

- python developed on 2.7 (also works on 3)
- pyserial 2.6 (or later)
- numpy
- scipy
- enum34

Other optional packages may be installed depending on your needs:

- sphinx and sphinx-bootstrap-theme (to build the doc)
- PyQt4 (for the graphical tools)
- bottle and tornado for REST API support and http-server

Once it is done, you can build and install pypot with the classical:

```
cd pypot
sudo python setup.py install
```

Testing your install You can test if the installation went well with:

```
python -c ``import pypot''
```

You will also have to install the driver for the USB2serial port. There are two devices that have been tested with pypot that could be used:

- USB2AX - this device is designed to manage TTL communication only
- USB2Dynamixel - this device can manage both TTL and RS485 communication.

On Windows and Mac, it will be necessary to download and install a FTDI (VCP) driver to run the USB2Dynamixel, you can find it here. Linux distributions should already come with an appropriate driver. The USB2AX device should not require a driver installation under MAC or Linux, it should already exist. For Windows XP, it should automatically install the correct driver.

> **note**
>
> On the side of the USB2Dynamixel there is a switch. This is used to select the bus you wish to communicate on. This means that you cannot control two different bus protocols at the same time.

On most Linux distributions you will not have the necessary permission to access the serial port. You can either run the command in sudo or better you can add yourself to the *dialout* or the *uucp* group (depending on your distribution):

```
sudo addgroup $USER dialout
sudo addgroup $USER uucp
```

At this point you should have a pypot ready to be used! In the extremely unlikely case where anything went wrong during the installation, please refer to the issue tracker.

Updating Currently, Pypot is still updating 'by hand', by command line while SSH into the robot.

If you are using PIP, enter:

```
pip install --upgrade pypot
```

If you are using the sources, go to the ~/dev/pypot folder and enter:

```
git pull
python setup.py install
```

Extending pypot

While pypot has been originally designed for controlling dynamixel based robots, it became rapidly obvious that it would be really useful to easily:

- control other types of motor (e.g. servo-motors controlled using PWM)
- control an entire robot composed of different types of motors (using dynamixel for the legs and smaller servo for the hands for instance)

While it was already possible to do such things in pypot, the library has been partially re-architectured in version 2.x to better reflect those possibilities and most importantly make it easier for contributors to add the layer needed for adding support for other types of motors.

> **note**
>
> While in most of this documentation, we will show how support for other motors can be added, similar methods can be applied to also support other sensors.

The rest of this section will describe the main concept behing pypot's architecture and then give examples of how to extend it.

Writing a new IO In pypot's architecture, the IO aims at providing convenient methods to access (read/write) value from a device - which could be a motor, a camera, or a simulator. It is the role of the IO to handle the communication:

- open/close the communication channel,
- encapsulate the protocol.

For example, the pypot.dynamixel.io.io.DxlIO (for dynamixel buses) open/closes the serial port and provides high-level methods for sending dynamixel packet - e.g. for getting a motor position. Similarly, writing the pypot.vrep.io.VrepIO consists in opening the communication socket to the V-REP simulator (thanks to V-REP's remote API) and then encapsulating all methods for getting/setting all the simulated motors registers.

> **warning**
>
> While this is not by any mean mandatory, it is often a good practice to write all IO access as synchronous calls. The higher-level synchronization loop is usually written as a pypot.robot.controller.AbstractController.

The IO should also handle the low-level communication errors. For instance, the pypot.dynamixel.io.io.DxlIO automatically handles the timeout error to prevent the whole communication to stop.

note

Once the new IO is written most of the integration into pypot should be done! To facilitate the integration of the new IO with the higher layer, we strongly recommend to take inspiration from the existing IO

- especially the pypot.dynamixel.io.io.DxlIO and the pypot.vrep.io.VrepIO ones.

Writing a new Controller A pypot.robot.controller is basically a synchronization loop which role is to keep up to date the state of the device and its "software" equivalent - through the associated IO.

In the case of the pypot.dynamixel.controller.DxlController, it runs a 50Hz loop which reads the actual position/speed/load of the real motor and sets it to the associated register in the pypot.dynamixel.motor.DxlMotor. It also reads the goal position/speed/load set in the pypot.dynamixel.motor.DxlMotor and sends them to the "real" motor.

As most controller will have the same general structure - i.e. calling a sync. method at a predefined frequency - pypot provides an abstract class, the pypot.robot.controller.AbstractController, which does exactly that. If your controller fits within this conception, you should only have to overide the pypot.robot.controller.AbstractController.update method.

In the case of the pypot.vrep.controller.VrepController, the update loop simply retrieves each motor's present position and send the new target position. A similar approach is used to retrieve values form V-REP sensors.

note

Each controller can run at its own pre-defined frequency and live within its own thread. Thus, the update never blocks the main thread and you can used tight synchronization loop where they are needed (e.g. for motor's command) and slower one when latency is not a big issue (e.g. a temperature sensor).

Integrate it into the Robot Once you have defined your Controller, you most likely want to define a convenient factory functions (such as pypot.robot.config.from_config or pypot.vrep.from_vrep) allowing users to easily instantiate their pypot.robot.robot.Robot with the new Controller.

By doing so you will permit them to seamlessly uses your interface with this new device without changing the high-level API. For instance, as both the pypot.dynamixel.controller.DxlController and the pypot.vrep.controller.VrepController only interact with the pypot.robot.robot.Robot through getting and setting values into pypot.robot.motor.Motor instances, they can be directly switch.

Quickstarts

QuickStart: discover and communicate with Dynamixel servomotors

Assume you have a Dynamixel servomotor connected to the computer, using a USB2AX or a USB2Dynamixel. The servomotor has to be powered by a SMPS2Dynamixel (or any other external power source), because the USB port does not deliver enough current.

Ports scan First we are going to discover all serial ports open on your computer. These ports (called 'COM' in Windows, '/dev/tty' in Linux) are used by serial devices (mainly USB devices usingserial communication), like a USB2AX, a Razor board...

```
import pypot.dynamixel

ports = pypot.dynamixel.get_available_ports()

if not ports:
```

```
    raise IOError(`no port found!')

print `ports found', ports
```

We start by importing the *dynamixel* (low-level) part of pypot. Then we use the pypot.dynamixel.get_available_ports function to create a list of all ports names.

Then we check if there is something in the port variable and raise an error if it's not the case.

The last line prints 'ports found' and the list of ports, which should result in something like (for Linux):

```
> ports found [''/dev/ttyACM0'', ``/dev/ttyUSB0'']
```

ID scan We use only one USB port, but we can plug several Dynamixel servomotors in serial. To be able to communicate with a given servo, we need to know its ID.

Never connect two servomotors with the same ID on the same bus !

We also have to know its baudrate (the frequency at which it talks) and its protocol (version 1 for MX and AX servos, version 2 for XL servos).

Its baudrate 57600 for a MX, 1000000 for a AX or XL and its ID should be 1.

Let start by defining the baudrate and protocol values (change them according to your setup):

```
using_XL320 = False
my_baudrate = 1000000
```

Then we scan the ports: for each port, for the 60 first IDs, we ask if a servomotor has this ID on this port (with the previously defined protocol and baudrate).

The IDs can theoretically go up to 254, but IDs above 60 are rarely used (ans the scanning takes some time).

```
for port in ports:
    print port
    try:
        if using_XL320:
            dxl_io = pypot.dynamixel.Dxl320IO(port, baudrate=my_baudrate)
        else:
            dxl_io = pypot.dynamixel.DxlIO(port, baudrate=my_baudrate)

        print ``scanning''
        found =  dxl_io.scan(range(60))
        print found
        dxl_io.close()
    except Exception, e:
        print e
```

You should get something like:

```
/dev/ttyACM0
scanning
[11, 12, 13, 14, 15]
/dev/ttyUSB0
scanning
[]
```

here, there are 5 servomotors connected on the /dev/ttyACM0 port and none one the '/dev/ttyUSB0' one.

Reading and writing in registers Dynamixel servomotors are called 'smart servos', because they are not controlled through a PWM signal as simple DC motors or servomotors, but they contain an electronical board with a memory. You write the goal position in a given register and the board controls the servomotor to reach the position. You can also write in the speed or max_torque registers. The full protocol is available on Robotis website

Let read the position of a servomotor and write a new goal position:

```
import pypot.dynamixel
import time

my_port = ``/dev/ttyACM0'' #Modify to fit your setup!
my_baudrate = 1000000   #Modify to fit your setup!
my_id = 11                        #Modify to fit your setup!

#start serial communication
dxl_io = pypot.dynamixel.DxlIO(my_port, baudrate=my_baudrate)

#get position of servo my_id
pos = dxl_io.get_present_position([my_id])
print pos

#allow the servomotor to move
dxl_io.enable_torque([my_id])

#set position of servo m_id to 90 degrees
dxl_io.set_goal_position({my_id: 90})

#wait a bit
time.sleep(2)

#put compliance back to the robot
dxl_io.disable_torque([my_id])

#end serial communication
dxl_io.close()
```

See pypot.dynamixel.io here to find all available registers functions. Remember that it is always easier to use the robot and motor abstractions.

QuickStart: create a Robot

What is a configuration file? In Pypot, there is a Robot object that contains the configuration of your robot: how many motors (with what IDs, on what port), their names, angle limits, and so on.

You can build a Robot object by hand, but it is much easier to launch a configuration from a configuration file. This text file contains a dictionnary, encoded in json.

The important fields are:

- **controllers** - This key holds the information pertaining to a controller and all the items connected to its bus.
- **motors** - This is a description of all the custom setup values for each motor. Meta information, such as the motor access name or orientation, is also included here. It is also there that you will set the angle limits of the motor.
- **motorgroups** - This is used to define alias of a group of motors (e.g. left_leg).

This is an example of a minimal config file:

```
{
  ``controllers'': {
    ``head_controller'': {
      ``sync_read'': true,
      ``attached_motors'': [
        ``head'',
      ],
      ``port'': ``auto''
    }
  },
  ``motorgroups'': {
    ``head'': [
      ``head_z'',
      ``head_y''
    ]
  },
  ``motors'': {
    ``head_y'': {
      ``offset'': 20.0,
      ``type'': ``AX-12'',
      ``id'': 37,
      ``angle_limit'': [
        -40,
        8
      ],
      ``orientation'': ``indirect''
    },
    ``head_z'': {
      ``offset'': 0.0,
      ``type'': ``AX-12'',
      ``id'': 36,
      ``angle_limit'': [
        -100,
        100
      ],
      ``orientation'': ``direct''
    }
  }
}
```

It contains:

- two **motors** called head_y and head_z
- one **motor group** called head and containing the head_y and head_z motors
- one **controller** called head_controller, which controls the motors of the motor group head. A controller is associated to a serial port and therefore a USB2serial device. Even if you can theoretically plug more than 100 servos on one port, it is unadvised (for electrical losses) to have more than 15.

Each robot-specific library (poppy-humanoid for example) contains its own configuration file.

See the robot object description for more details on the contents of the configuration file.

Test the Ergo Jr configuration For test purposes, Pypot also contains a poppy Ergo Jr configuration, ready to use as a Python dictionnary.

```
from pypot.robot.config import ergo_robot_config
import pypot.robot

my_config = dict(ergo_robot_config)
print my_config[''controllers'']
print my_config[''motorgroups'']
print my_config[''motors'']
```

If you convert this dictionnary to a robot, pypot checks if the motors are connected and tells you which motors it can't find. If all the motors are found, you can directly access the motors using their names and get and set their registers directly using their names:

```
ergo_robot = pypot.robot.from_config(my_config)
print ergo_robot.m2.present_position #get register present_position of motor m2

ergo_robot.m6.compliant = False       #enable torque of the m6 motor
ergo_robot.m6.goal_position = 20      #set goal position of motor m6 to 20 degree

time.sleep(2)                                    #wait for the robot to move

ergo_robot.m6.compliant = True       #disable torque of the m6 motor
time.sleep(0.1)
```

Compliance is the fact that a motor can be moved by hand, without resisting. In order to have the robot move by itself, you should first set the compliance to False.

> **warning**
>
> Removing compliance will allow the motors to move to their goal_position, resulting in maybe sudden moves in the robot. Be sure to let it enough space to move.
>
> **note**
>
> Remark the time.sleep(0.1) in the last line: at the end of the script, the serial connection is closed and, if you don't wait a little bit, the connection may close before the last order (here: ergo_robot.m6.compliant = True) is sent.

See the discover quickstart or the motor object description for more precisions on how to control a robot

QuickStart: use a primitive

What is a primitive? A primitive is a behavior, simple or complex, that can be started, stopped and executes in parallel of your script.

Primitives allow you to build really easily complex and modular behavior. For example, you can have one primitive to move the head, one to walk and one to check if someone moves your arm.

A primitive must first be created, then you call the *start()* function to start it. When you want to stop it, simply use the *stop()* function. While the primitive runs, it call the *run()* function in a loop.

Some primitives are defined in Pypot. Other of specific to a robot and therefore defined in the specific libraries. You can of course define you own primitives.

Example: Record and replay moves The pypot.primitive.move module contains utility classes to help you record and play moves. A pypot.primitive.move.Move object simply contains a sequence of positions.

The pypot.primitive.move.MoveRecorder and pypot.primitive.move.MovePlayer are primitives included in Pypot and allow you to record a move and replay it.

Assuming you have a pypot.robot.robot.Robot poppy already created:

```
from pypot.primitive.move import Move, MoveRecorder, MovePlayer

record_frequency = 20.0 # This means that a new position will be recorded 50 times per seco
recorded_motors = poppy.motors # We will record the position of all motors

#disable torque for all motors
for m in recorded_motors:
    m.compliant = True

#create the recorder primitive
recorder = MoveRecorder(poppy, record_frequency, recorded_motors)

print ``start recording''
recorder.start()

time.sleep(20)

print ``stop recording''
recorder.stop()

#save the recorded move in a file
with open(`mymove.json', `w') as f:
    recorder.move.save(f)
```

During the 20 seconds wait, move the robot. The angular positions will be recorded, stored in recorder.move and, at the end, saved into a file called mymove.json. You can open this file with text editor to check what it contains.

We can now replay the move:

```
#read the file containing the move
with open(`mymove.json') as f:
    loaded_move = Move.load(f)

#create the primitive
player = MovePlayer(poppy, loaded_move)

#enable torque for all motors
for m in poppy.motors:
    m.compliant = False

print ``starting''
player.start()
player.wait_to_stop()
print ``finished''

#disable torque for all motors
for m in poppy.motors:
    m.compliant = True

time.sleep(0.1)
```

To learn more about primitives, see here.

Pypot in details

Dynamixel Low-level IO

The low-level API almost directly encapsulates the communication protocol used by dynamixel motors. More precisely, this class can be used to:

- open/close the communication
- discover motors (ping or scan)
- access the different registers (read and write)

The communication is thread-safe to avoid collision in the communication buses.

All servomotors on a bus must have a unique ID and the same baudrate.

> **note**
>
> All new servos have ID 1, so when using new motors, it is strongly advise to first plug them one by one and change their IDs

The pypot.dynamixel.io.io.DxlIO class is used to handle the communication with a particular port using the version 1 of robotis protocol, allowing to communicate with Dynamixel MX and AX servomotors.

The pypot.dynamixel.io.io_320.Dxl320IO class is used to handle the communication with a particular port using the version 2 of robotis protocol, allowing to communicate with Dynamixel XL servomotors.

> **note**
>
> A port can only be accessed by a single DxlIO or Dxl320IO instance. Therefore it is impossible to control XL servomotors on the same bus as MX or AX servomotors.

> **note**
>
> Pypot uses the TTL (3-pin) protocol. Therefore, it is not compatible with R Dynamixel servomotors (RX-28, MX-28R...).

Finding ports The pypot.dynamixel module offers several useful function to learn the state of your connections:

- pypot.dynamixel.get_available_ports discovers the open serial ports. An optionnal argument allows you to ask only for free (not used by a serial connection) ports.
- pypot.dynamixel.find_port takes a list of motors IDs and scan the available ports until it finds the motors.
- pypot.dynamixel.autodetect_robot scans all ports and creates a pypot.robot.robot.Robot out of the found motors

```
import pypot.dynamixel

ports = pypot.dynamixel.get_available_ports()
if not ports:
    raise IOError(`no port found!')
print(`ports found', ports)
```

Opening/Closing a communication port, scanning motors Once you know the name of the port you want to connect to, you can open a connection through a virtual communication port to your device:

```
dxl_io = pypot.dynamixel.DxlIO(portName, baudrate=57600)
```

The portName is a string and the baudrate argument is optionnal and defaults at 1000000.

note

Default baudrate for MX servomotors is 57600, but this should be changed to 1000000 while building the robot.

To detect the motors and find their id you can scan the bus:

```
dxl_io.scan()
>>> [4, 23, 24, 25]
```

This should produce a list of the ids of the motors that are connected to the bus.

To avoid spending a long time searching all possible values, you can add a list of values to test:

```
dxl_io.scan([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> [4]
```

Or, you can use the shorthand:

```
dxl_io.scan(range(10))
>>> [4]
```

The communication can be closed using the ~pypot.dynamixel.io.DxlIO.close method:

```
dxl_io.close()
```

note

The class pypot.dynamixel.io.io.DxlIO can also be used as a Context Manager (the pypot.dynamixel.io.io.DxlIO.close method will automatically be called at the end). : For instance:

with pypot.dynamixel.DxlIO('/dev/ttyUSB0') as dxl_io: : ...

Registers access

note

Appart from the initial parametrization of your motors, you should not use these low level function but instead the equivalent access provided in _controller.

Now you have the id of the connected motors, you can access their registers. Try to find out the present position (in degrees) of the motor with ID 4:

```
dxl_io.get_present_position([4])
>>> (67.8, )
```

The motors are handled in degrees where 0 is considered the central point of the motor turn. For the MX motors, the end points are -180° and 180°. For the AX and RX motors, these end points are -150° to 150°.

You can also write a goal position (in degrees) to the motor using the following:

```
dxl_io.set_goal_position({4: 0})
```

As you can see on the example above, you should always pass the id parameter as a list. This is intended as getting a value from several motors takes the same time as getting a value from a single motor (thanks to the SYNC_READ instruction). Similarly, we use dictionary with pairs of (id, value) to set value to a specific register of motors and benefit from the SYNC_WRITE instruction. The equivalent instructions for several motors would be:

```
dxl_io.get_present_position([4, 5, 6])
>>> (67.8, -12.6, 23.8)
dxl_io.set_goal_position({4: 0, 5 : 10, 6 : -25})
```

Registers in Dynamixel servomotors allow you to (among others): - Read the current position - Read and write the goal position - Read the current speed - Read and write the goal speed - Read and write the control mode ('joint' for a standard servomotor, 'wheel' for a DC motor equivalent: can turn forever, speed control only) - Read and write angle

limits: if you ask a servo to go beyond a limit, it will stop at the limit - Read and write the maximum torque (between 0 and 100)

The list of all functions is available in pypot.dynamixel.io.io.DxlIO. The syntax is the same for all registers: all the getter functions takes a list of ids as argument and the setter takes a dictionary of (id: value) pairs.

Set parameters for Poppy robots This code can be used to initialize a new motor to the right ID, baudrate and return delay. be sure to adapt it to your configuration!

```
import pypot.dynamixel, time

ports = pypot.dynamixel.get_available_ports()
if not ports:
    raise IOError(`no port found!')
print `ports found', ports

my_port = ports[0]

old_id = 1           #Should be 1 if the motor has never been configured\n'',
new_id = 11            #Change this value\n'',

old_baudrate = 57600   #Should be 57600 for new MX-28 or MX-64, 1000000 for new AX612A or X
new_baudrate = 1000000 #Should be 1000000''

dxl_io = pypot.dynamixel.DxlIO(my_port, baudrate=old_baudrate)

found =  dxl_io.scan([old_id])
if old_id in found:
    dxl_io.set_return_delay_time({old_id : 0})
    dxl_io.change_id({old_id : new_id})
    dxl_io.change_baudrate({new_id : new_baudrate})

else:
    print ``motor '',old_id,'' not found on port '', my_port, '' at baudrate '',old_baudrat

dxl_io.close()

time.sleep(2)

dxl_io = pypot.dynamixel.DxlIO(my_port, baudrate=new_baudrate)
found =  dxl_io.scan([new_id])
if new_id in found:
    print ``success''
else:
    print ``motor '',new_id,'' not found on port '', my_port, '' at baudrate '',new_baudrat
```

All motors work sufficiently well with a 12V supply. Some motors can use more than 12V but you must be careful not to connect an 18V supply on a bus that contains motors that can only use 12V! Connect this 12V SMPS supply (switch mode power supply) to a Robotis SMPS2Dynamixel device which regulates the voltage coming from the SMPS. Connect your controller device and a single motor to this SMPS2Dynamixel.

The Robot object and the controllers

Using the robot abstraction While the low_level provides access to all functionalities of the dynamixel motors, it forces you to have one communication call for function call, which can take a non-negligible amount of time. In particular, most programs will need to have a really fast read/write synchronization loop, where we typically read all motor position, speed, load and set new values, while in parallel we would like to have higher level code that computes those new values.

This is pretty much what the robot abstraction is doing for you. More precisely, through the use of the class pypot.robot.robot.Robot you can:

- automatically initialize all connections (make transparent the use of multiple USB2serial connections),
- define pypot.dynamixel.motor.DxlMotor.offset and pypot.dynamixel.motor.DxlMotor.direct attributes for motors,
- automatically define accessor for motors and their most frequently used registers (such as pypot.dynamixel.motor.DxlMotor.goal_position, pypot.dynamixel.motor.DxlMotor.present_speed, pypot.dynamixel.motor.DxlMotor.present_load, pypot.dynamixel.motor.DxlMXMotor.pid, pypot.dynamixel.motor.DxlMotor.compliant),
- define read/write synchronization loop that will run in background.

We will first see how to define your robot thanks to the writing of a configuration, then we will describe how to set up synchronization loops. Finally, we will show how to easily control this robot through asynchronous commands.

Create the robot object

The configuration file See this quickstart for a quick intro on how to use the configuration files.

The configuration, described as a Python dictionary, contains several important features that help build both your robot and the software to manage you robot. The configuration can also be loaded from any file that can be loaded as a dict (e.g. a JSON file).

The important fields are:

1. **controllers**: This key holds the information pertaining to a controller and all the items connected to its bus. You can have a single or multiple pypot.dynamixel.controller.DxlController. : You must specify the attached motors (or motor groups) and port that the device is connected to, or "auto". When loading the configuration, pypot will automatically try to find the port with the corresponding attached motor ids. You also have to specify the protocol (1 or 2) and you you want to use the SYNC_READ instruction (see below). :

```
my_config['controllers'] = {}
my_config['controllers']['upper_body_controler'] = {
    `port': `/dev/ttyUSB0',
    `sync_read': False,
    `attached_motors': ['torso', `head', `arms'],
    `protocol': 1,
}
```

2. **motorgroups**: These defines the different motors group corresponding to the structure of your robot. It will automatically create an alias for the group. Groups can be nested, i.e. a group can be included inside another group, as in the example below:

```
my_config['motorgroups'] = {
    `torso': ['arms', `head_x', `head_y'],
    `arms': ['left_arm', `right_arm'],
    `left_arm': ['l_shoulder_x', `l_shoulder_y', `l_elbow'],
```

```
    `right_arm': ['r_shoulder_x', `r_shoulder_y', `r_elbow']
}
```

3. **motors**: This is a description of all the custom setup values for each motor:

```
my_config['motors'] = {}
my_config['motors']['l_hip_y'] = {
    `id': 11,
    `type': `MX-28',
    `orientation': `direct',
    `offset': 0.0,
    `angle_limit': (-90.0, 90.0),
}
```

Mandatory information for each motor is:

- Motor name and ID
- Motor type: 'MX-28', 'MX-64', 'AX-12A' or 'XL-320'. This will change which attributes are available (e.g. compliance margin versus pid gains).
- Limit angles
- Orientation: describes whether the motor will act in an anti-clockwise fashion (direct) or clockwise (indirect) when asked to increase the angle.
- Offset: describe the offset between physical zero of the servo and 'software zero', the position of the motor when requested to go at angle 0.

In the source of pypot.robot.config, you can find the configuration dictionnary of a Poppy Ergo Jr robot.

Use the configuration To create a pypot.robot.robot.Robot object from a Python dictionnary, use the pypot.robot.config.from_config function function:

```
import pypot.robot

robot = pypot.robot.from_config(my_config)
```

You can also create a pypot.robot.robot.Robot by detecting the available Dynamixel servomotors:

```
from pypot.dynamixel import autodetect_robot

my_robot = autodetect_robot()
```

To save your configuration as a json file, use the following code:

```
import json

config = my_robot.to_config()

with open(`myconfig.json', `w') as f:
    json.dump(config, f, indent=2)
```

If you have your configuration in a json file, here is how to open it:

```
import json
import pypot.robot

ergo = pypot.robot.from_json(`ergo.json')
```

While having the configuration as a file is convenient to share the same config on multiple machine, it also slows the creation of the pypot.robot.robot.Robot.

Dynamixel controller and Synchronization Loop The pypot.robot.robot.Robot held instances of pypot.dynamixel.motor.DxlMotor. Each of these instances represents a real motor of your physical robot. The attributes of those "software" motors are automatically synchronized with the real "hardware" motors. In order to do that, the pypot.robot.robot.Robot class uses a pypot.dynamixel.controller.DxlController which defines synchronization loops that will read/write the registers of dynamixel motors at a predefined frequency.

> **warning**
>
> The synchronization loops will try to run at the defined frequency, however don't forget that you are limited by the bus bandwidth! For instance, depending on your robot you will not be able to read/write the position of all motors at 100Hz. Moreover, the loops are implemented as python thread and we can thus not guarantee the exact frequency of the loop.

By default the class pypot.robot.robot.Robot uses a particular controller pypot.dynamixel.syncloop.BaseDxlController which already defines synchronization loops. More precisely, this controller:

- reads the present position, speed, load at 50Hz,
- writes the goal position, moving speed and torque limit at 50Hz,
- writes the pid or compliance margin/slope (depending on the type of motor) at 10Hz,
- reads the present temperature and voltage at 1Hz.

So, in most case you should not have to worry about synchronization loop and it should directly work.

The synchronization loops are automatically started when instantiating your robot if you set the sync_read parameter of your controller to True. Otherwise, start it with the method pypot.robot.robot.Robot.start_sync. You can also stop the synchronization if needed (see the pypot.robot.robot.Robot.stop_sync method).

> **note**
>
> With the current version of pypot, you can not yet indicate in the configuration which subclasses of pypot.dynamixel.controller.DxlController you want to use. If you want to use your own controller, you should either modify the config parser, modify the pypot.dynamixel.syncloop.BaseDxlController class or directly instantiate the pypot.robot.robot.Robot class.

> **warning**
>
> You should never set values to motors when the synchronization is not running.

Now you have a robot that is reading and writing values to each motor in an infinite loop. Whenever you access these values, you are accessing only their most recent versions that have been read at the frequency of the loop. This automatically make the synchronization loop run in background. You do not need to wait the answer of a read command to access data (this can take some time) so that algorithms with heavy computation do not encounter a bottleneck when values from motors must be known.

Now you are ready to create some behaviors for your robot.

Controlling your robot

Robot overview The main fields of the pypot.robot.robot.Robot are:

- motors: list of pypot.dynamixel.motor.DxlMotor. Example: list all motors:

  ```
  for m in robot.motors:
      print m.name
  ```

  Each motor name is a field of the robot, so you can control a motor directly:

  ```
  print robot.head_z.present_position
  ```

  Each motor group is also a field:

```
for m in robot.head:
    print m.name
```

- compliant: This is a shortcut to set all motors compliance to the same value in one command:

  ```
  robot.compliant = True
  ```

- primitives: You can attach primitives to a robot and this field lists them.
- active_primitives: from above primitives, which are currently running
- sensors: list of available sensors, work in progress

Some useful functions of the pypot.robot.robot.Robot class:

- pypot.robot.robot.Robot.power_up to set maximum torque and remove compliance.

Synchronized moves The pypot.dynamixel.motor.DxlMotor allows you to control motors in position and speed, but, at the pypot.robot.robot.Robot level, you can give orders to a set of motors to make a synchronized move using the pypot.robot.robot.Robot.goto_position function.

This is especially useful for choregraphies, because the pypot.robot.robot.Robot.goto_position function ensures that all motors smoothly reach their final positions at the same time, while using the goal_position field will lead all motors to go to the same speed, without time synchronization.

For example to move the head to angles (0, 20.) degrees in 3 seconds:

```
robot.goto_position({``head_z'':0. , ``head_y'':20}, 3)
```

By default, this function return immediatelly and is cancelled if another one is run later, even if the 3 seconds are not over.

You can set the optionnal *wait* parameter to True to make this function blocking, therefore the next line in the script will execute only when the 3 seconds are over.

The other optionnal parameter is *control*. You can specify 'dummy' or 'minijerk' (default) to define which algorithm is used in used in background to bring the motor to the desired position.

'dummy' is a simple controller, where you divide the angle to travel by the time and you set the goal speed to this value. As the motor can't go from thois speed to 0 at arrival in no time, a slight overshoot can happen. The 'minijerk' controller has a more complex algorithm to slow down before and arrive on time without overshoot.

```
robot.goto_position({``head_z'':0. , ``head_y'':20}, 3, control='dummy', wait=True)
```

Closing the robot To make sure that everything gets cleaned correctly after you are done using your pypot.robot.robot.Robot, you should always call the pypot.robot.robot.Robot.close method. Doing so will ensure that all the controllers attached to this robot, and their associated dynamixel serial connection, are correctly stopped and cleaned.

It is advised to use the contextlib.closing decorator to make sure that the close function of your robot is always called whatever happened inside your code:

```
from contextlib import closing

import pypot.robot

# The closing decorator make sure that the close function will be called
# on the object passed as argument when the with block is exited.

with closing(pypot.robot.from_json(`myconfig.json')) as my_robot:
    # do stuff without having to make sure not to forget to close my_robot!
```

```
pass
```

note

Note calling the pypot.robot.robot.Robot.close method on a pypot.robot.robot.Robot can prevent you from opening it again without terminating your current Python session. Indeed, as the destruction of object is handled by the garbage collector, there is no mechanism which guarantee that we can automatically clean it when destroyed.

When closing the robot, we also send a stop signal to all the primitives running and wait for them to terminate. See section primitives for details on what we call primitives.

warning

You should be careful that all your primitives correctly respond to the stop signal. Indeed, having a blocking primitive will prevent the pypot.robot.robot.Robot.close method to terminate (please refer to primitives for details).

The Motor object

Overview The pypot.robot.robot.Robot class contains a list of pypot.dynamixel.motor.DxlMotor, each pypot.dynamixel.motor.DxlMotor being linked to a physical Dynamixel motor.

Registers This class provides access to (see pypot.dynamixel.motor.DxlMotor.registers for an exhaustive list):

- id: motor id
- motor name
- motor model
- present position/speed/load
- goal position/speed/load
- compliant
- motor orientation and offset
- angle limit
- temperature
- voltage

Temperature and load can give you an idea of how much effort a motor produces:

```
for m in robot.motors:
    print ``motor '',m.name, ``('',m.model,''), id: '',m.id
    print `temperature: `,m.temperature
    print ``load: '', m.load
```

Torque and compliance A motor with compliance will not exert any torque. You can move it by hand. Removing compliance will cause the motor to exert torque and therefore move to get to its goal position. The compliance can be set for each motor individually:

```
robot.head_z.compliant = False
```

or can be set at robot level.

You can set the compliance mode to 'safe' (as opposed to 'dummy') to have the robot set compliance at angle limits, preventing you to move it outside of the authorized range. It also prevents the brutal moves when putting back compliance and the robot wants to go back inside its angle limits.

```
print robot.head_z.angle_limit
robot.head_z.compliant_behavior(`safe')
robot.head_z.compliant = True
#now try moving the head with your hands beyond the angle limits
```

You can also change the maximal torque that the robot can use. Use a value between 0 (no torque) and 100 (max torque). The resulting maximal torque depends onthe model of the robot:

```
robot.head_z.max_torque = 20
```

Reducing the maximal torque makes your robot less powerful and therefore less harmfull and adds elasticity to the joints: if you apply a force on it,it moves because it does not have enough torque to resist, but it goes back to its goal position when you stop applying the force.

Controlling motors

Controlling in position Dynamixel servomotors will use their internal controller to reach and stay at the angle defined in the goal_position register (in degree). If this angle isn't between the minimum and maximum angle (see pypot.dynamixel.motor.DxlMotor.angle_limit), the goal angle will automatically be taken back into the limits:

```
for m in robot.motors:
    m.compliant = False
    m.goal_position = 0

time.sleep(2)

for m in robot.motors:
    print ``position: '', m.present_position, ``(limits: '',m.angle_limit, '')''
```

Controlling in speed You can also control your robot in speed. Set the pypot.dynamixel.motor.DxlMotor.goal_speed attribute to the desired value in degree per seconds. This automatically sets the pypot.dynamixel.motor.DxlMotor.goal_position to the maximal or minimal value (depending on the sign of the speed).

The motor will remain at the given speed until it gets a new order or it reaches its angle limit.

Example of robot making 'yes' with its head:

```
goal = 20

t_init = time.time()

while time.time() - t_init < 20:

    if abs(robot.head_y.present_position) > abs(goal):
        goal = -goal

    speed = 0.1*(goal - robot.head_y.present_position)
    robot.head_y.goal_speed = speed

    time.sleep(0.1)
```

note

You could also use the wheel mode settings where you can directly change the pypot.dynamixel.motor.DxlMotor.moving_speed. Nevertheless, while the motor will turn infinitely with the wheel mode, here with the pypot.dynamixel.motor.DxlMotor.goal_speed the motor will still respect the angle limits.

warning

If you set both pypot.dynamixel.motor.DxlMotor.goal_speed and pypot.dynamixel.motor.DxlMotor.goal_position only the last command will be executed. Unless you know what you are doing, you should avoid to mix these both approaches.

The pypot.dynamixel.motor.DxlMotor.goto_position function If you want a servo to go to a certain position in a certain time (for synchronization reasons...), use the pypot.dynamixel.motor.DxlMotor.goto_position function. It take two mandatory arguments: position to reach (in degrees) and the duration:

```
robot.head_z.goto_position(20, 3)
```

To synchronize several motors, have a look at the pypot.robot.robot.Robot.goto_position at robot level.

By default, this function return immediatelly and is cancelled if another one is run later, even if the 3 seconds are not over.

You can set the optionnal *wait* parameter to True to make this function blocking, therefore the next line in the script will execute only when the 3 seconds are over.

The other optionnal parameter is *control*. You can specify 'dummy' or 'minijerk' (default) to define which algorithm is used in used in background to bring the motor to the desired position.

'dummy' is a simple controller, where you divide the angle to travel by the time and you set the goal speed to this value. As the motor can't go from thois speed to 0 at arrival in no time, a slight overshoot can happen. The 'minijerk' controller has a more complex algorithm to slow down before and arrive on time without overshoot.

```
robot.head_z.goto_position(20, 3, control='dummy', wait=True)
```

TODO: sensors doc, but work in progress

Primitives

In the previous sections, we have shown how to make a simple behavior thanks to the pypot.robot.robot.Robot abstraction. pypot.primitive.primitive.Primitive allows you to create more complexe, parallelized behavior really easily.

What is a "Primitive"? We call pypot.primitive.primitive.Primitive any simple or complex behavior applied to a pypot.robot.robot.Robot. A primitive can access all sensors and effectors in the robot. It is started in a thread and can therefore run in parallel with other primitives.

All primitives implement the pypot.primitive.primitive.Primitive.start, pypot.primitive.primitive.Primitive.stop, pypot.utils.stoppablethread.StoppableThread.pause and pypot.utils.stoppablethread.StoppableThread.resume. Unlike regular python thread, primitive can be restart by calling again the pypot.primitive.primitive.Primitive.start method.

To check if a primitive is finished, use the pypot.primitive.primitive.Primitive.start.is_alive method (will output True if primitive is paused but False if it has been stopped or if it's finished).

The pypot.primitive.primitive.PrimitiveLoop is a pypot.primitive.primitive.Primitive that repeats its behavior forever.

A primitive is supposed to be independent of other primitives. In particular, a primitive is not aware of the other primitives running on the robot at the same time.

This is really important when you create complex behaviors - such as balance - where many primitives are needed. Adding another primitive - such as walking - should be direct and not force you to rewrite everything. Furthermore, the balance primitive could also be combined with another behavior - such as shoot a ball - without modifying it.

Primitive manager To ensure this independence, the primitive is running in a sort of sandbox. More precisely, this means that the primitive has not direct access to the robot. It can only request commands (e.g. set a new goal position of a motor) to a pypot.primitive.manager.PrimitiveManager which transmits them to the "real" robot. As multiple primitives can run on the robot at the same time, their request orders are combined by the manager.

The primitives all share the same manager. In further versions, we would like to move from this linear combination of all primitives to a hierarchical structure and have different layer of managers.

The manager uses a filter function to combine all orders sent by primitives. By default, this filter function is a simple mean but you can choose your own specific filter (e.g. add function).

> **warning**
>
> You should not mix control through primitives and direct control through the pypot.robot.robot.Robot. Indeed, the primitive manager will overwrite your orders at its refresh frequency: i.e. it will look like only the commands send through primitives will be taken into account.

Default primitives An example on how to run a primitive is shown here.

Another primitive provided with Pypot is the pypot.primitive.utils.Sinus one. It allows you to apply a sinusoidal move of a given frequency and intensity to a motor or a list of motors:

```
from pypot.primitive.utils import Sinus

sinus_prim_z = Sinus(poppy, 50, ``head_z'', amp=30, freq=0.5, offset=0, phase=0)

#set phase to 90° to have the y sinus 1/4 of phase late compared to the z sinus
sinus_prim_y = Sinus(poppy, 50, ``head_y'', amp=20, freq=1, offset=0, phase=90)

print ``start moving''
sinus_prim_z.start()
sinus_prim_y.start()

time.sleep(20)

print ``pause move''
sinus_prim_z.pause()
sinus_prim_y.pause()

time.sleep(5)

print ``restart move''
sinus_prim_z.resume()
sinus_prim_y.resume()

time.sleep(20)

print ``stop moving''
sinus_prim_z.stop()
sinus_prim_y.stop()
```

Other default primitives are:

- pypot.primitive.utils.Cosinus for a cosinus move
- pypot.primitive.utils.Square for a square (go to max position, wait duty*cycle time, go to minposition, wait (1 - duty)*cycle time)
- pypot.primitive.utils.PositionWatcher: records and saves all positions of the given motors. You have a plot function to see your data.
- pypot.primitive.utils.SimplePosture: you should define a target_position as a dictionnary and the robot will go to this position

Writing your own primitive To write you own primitive, you have to subclass the pypot.primitive.primitive.Primitive class. It provides you with basic mechanisms (e.g. connection to the manager, setup of the thread) to allow you to directly "plug" your primitive to your robot and run it.

Important instructions When writing your own primitive, you should always keep in mind that you should never directly pass the robot or its motors as argument and access them directly. You have to access them through the self.robot and self.robot.motors properties.

Indeed, at instantiation the pypot.robot.robot.Robot (resp. pypot.dynamixel.motor.DxlMotor) instance is transformed into a pypot.primitive.primitive.MockupRobot (resp. pypot.primitive.primitive.MockupMotor). Those class are used to intercept the orders sent and forward them to the pypot.primitive.manager.PrimitiveManager which will combine them. By directly accessing the "real" motor or robot you circumvent this mechanism and break the sandboxing.

If you have to specify a list of motors to your primitive (e.g. apply the sinusoid primitive to the specified motors), you should either give the motors name and access the motors within the primitive or transform the list of pypot.dynamixel.motor.DxlMotor into pypot.primitive.primitive.MockupMotor thanks to the pypot.primitive.primitive.Primitive.get_mockup_motor method. For instance:

```
class MyDummyPrimitive(pypot.primitive.Primitive):
    def run(self, motors_name):
        motors = [getattr(self.robot, name) for name in motors_name]

        while True:
            for m in motors:
                ...
```

or:

```
class MyDummyPrimitive(pypot.primitive.Primitive):
    def run(self, motors):
        fake_motors = [self.get_mockup_motor(m) for m in motors]

        while True:
            for m in fake_motors:
                ...
```

When overriding the pypot.primitive.primitive.Primitive, you are responsible for correctly handling those events. For instance, the stop method will only trigger the should stop event that you should watch in your run loop and break it when the event is set. In particular, you should check the pypot.utils.stoppablethread.StoppableThread.should_stop and pypot.utils.stoppablethread.StoppableThread.should_pause in your run loop. You can also use the pypot.utils.stoppablethread.StoppableThread.wait_to_stop and pypot.utils.stoppablethread.StoppableThread.wait_to_resume to wait until the commands have really been executed.

note

You can refer to the source code of the pypot.primitive.primitive.LoopPrimitive for an example of how to correctly handle all these events.

Examples As an example, let's write a simple primitive that recreate the dance behavior written in the dance_ section. Notice that to pass arguments to your primitive, you have to override the pypot.primitive.primitive.Primitive.__init__ method.

> **note**
>
> You should always call the super constructor if you override the pypot.primitive.primitive.Primitive.__init__ method.

```
import time

import pypot.primitive
class DancePrimitive(pypot.primitive.Primitive):

    def __init__(self, robot, amp=30, freq=0.5):
        self.robot = robot
        self.amp = amp
        self.freq = freq
        pypot.primitive.Primitive.__init__(self, robot)

    def run(self):
        amp = self.amp
        freq = self.freq
        # self.elapsed_time gives you the time (in s) since the primitive has been running
        while self.elapsed_time < 30:
            x = amp * numpy.sin(2 * numpy.pi * freq * self.elapsed_time)

            self.robot.base_pan.goal_position = x
            self.robot.head_pan.goal_position = -x

            time.sleep(0.02)
```

To run this primitive on your robot, you simply have to do:

```
ergo_robot = pypot.robot.from_config(...)

dance = DancePrimitive(ergo_robot,amp=60, freq=0.6)
dance.start()
```

If you want to make the dance primitive infinite you can use the pypot.primitive.primitive.LoopPrimitive class:

```
class LoopDancePrimitive(pypot.primitive.LoopPrimitive):
    def __init__(self, robot, refresh_freq, amp=30, freq=0.5):
        self.robot = robot
        self.amp = amp
        self.freq = freq
        LoopPrimitive.__init__(self, robot, refresh_freq)

    # The update function is automatically called at the frequency given on the constructo
    def update(self):
        amp = self.amp
        freq = self.freq
        x = amp * numpy.sin(2 * numpy.pi * freq * self.elapsed_time)

        self.robot.base_pan.goal_position = x
        self.robot.head_pan.goal_position = -x
```

And then run it with:

```
ergo_robot = pypot.robot.from_config(...)

dance = LoopDancePrimitive(ergo_robot, 50, amp = 40, freq = 0.3)
# The robot will dance until you call dance.stop()
dance.start()
```

Attaching a primitive to the robot You can also attach a primitive to the robot (at start up for example) and then use it more easily.

For example with our DancePrimitive:

```
ergo_robot = pypot.robot.from_config(...)

ergo_robot.attach_primitive(DancePrimitive(ergo_robot), `dance')
ergo_robot.dance.start()
```

By attaching a primitive to the robot, you make it accessible from within other primitive:

```
class SelectorPrimitive(pypot.primitive.Primitive):
    def run(self):
        if song == `my_favorite_song_to_dance' and not self.robot.dance.is_alive():
            self.robot.dance.start()
```

note

In this case, instantiating the DancePrimitive within the SelectorPrimitive would be another solution.

Other useful features

Using a simulated robot with V-REP

See here for a quickstart using Poppy Humanoid.

What is V-rep ? As it is often easier to work in simulation rather than with the real robot, pypot has been linked with the V-REP simulator. It is described as the "Swiss army knife among robot simulators" and is a very powerful tool to quickly (re)create robotics setup. As presenting V-REP is way beyond the scope of this tutorial, we will here assume that you are already familiar with this tool. Otherwise, you should directly refer to V-REP documentation.

Details about how to connect pypot and V-REP can be found in this post.

The connection between pypot and V-REP was designed to let you seamlessly switch from your real robot to the simulated one. It is based on V-REP's remote API.

In order to connect to V-REP through pypot, you will only need to install the V-REP simulator. Pypot comes with a specific pypot.vrep.io.VrepIO designed to communicate with V-REP through its remote API.

This IO can be used to:

- connect to the V-REP server : pypot.vrep.io.VrepIO
- load a scene : pypot.vrep.io.VrepIO.load_scene
- start/stop/restart a simulation : pypot.vrep.io.VrepIO.start_simulation, pypot.vrep.io.VrepIO.stop_simulation, pypot.vrep.io.VrepIO.restart_simulation
- pause/resume the simulation : pypot.vrep.io.VrepIO.pause_simulation pypot.vrep.io.VrepIO.resume_simulation
- get/set a motor position : pypot.vrep.io.VrepIO.get_motor_position, pypot.vrep.io.VrepIO.set_motor_position

- get an object position/orientation : pypot.vrep.io.VrepIO.get_object_position, pypot.vrep.io.VrepIO.get_object_orientation

Connecting to V-REP First launch V-rep.

Then create your pypot.robot.robot.Robot using the pypot.vrep.from_vrep method, providing the vrep host, port and scene instead of the pypot.robot.config.from_config method:

```
# Working with the simulated version
import pypot.vrep

poppy = pypot.vrep.from_vrep(config, vrep_host, vrep_port, vrep_scene)
```

This function tries to connect to a V-REP instance and expects to find motors with names corresponding as the ones found in the config.

> **note**
>
> The returned pypot.robot.robot.Robot will also provide a convenient reset_simulation method which resets the simulation and the robot position to its intial stance.

Default host is localhost ('127.0.0.1') and default port is 19997 (the default one for V-rep), so if you launched V-rep on the same computer, you can use the pypot.vrep.from_vrep function with only the config argument.

The default vrep_scene is None, you can set it to the path of any .ttt file (containing, for example, additionnal objects that your robot can interact with).

The tracked_objects optionnal parameter is a list of V-REP dummy object to track, while the tracked_collisions is a list of V-REP collision to track.

The pypot.robot.robot.Robot object will be equivalent to the one created in the case of a real robot, but not all dynamixel registers have their V-REP equivalent. For the moment, only the control of the position is used. Primitives (that use only thr mootrs positions) can be used without problems.

If you use a creature-specific library to create your pypot.robot.robot.Robot (as poppy-humanoid for example), you can simply use the 'simulator' argument. If V-rep is not on the same machine, specify the vrep_host and vrep_port arguments:

```
from poppy_humanoid import PoppyHumanoid

poppy = PoppyHumanoid(simulator='vrep')
```

REST API

Any pypot.robot.robot.Robot object can be to remotely accessed and controled through TCP network.

This can be useful to:

- separate the low-level control running on an embedded computer and higher-level computation on a more powerful computer
- control your Poppy robot from any language (C++, javascript...) able to use tcp sockets
- remote control your robot without having to install all Poppy libraries

The protocol, described here, allows the access of all the robot variables and method (including motors and primitives) via a JSON request. Two transport methods have been developed so far:

- HTTP via GET and POST request (see the pypot.server.httpserver.HTTPRobotServer)
- ZMQ socket (see the pypot.server.zmqserver.ZMQRobotServer)

The pypot.server.rest.RESTRobot has been abstracted from the server, so you can easily add new transport methods if needed.

> **note**
>
> A third server is available in Pypot: pypot.server.snap.SnapRobotServer allows you to run Snap! directly on the robot.

ZMQ method This method is quick and powerful but needs the pyzmq library installed.

Server-side code (launch on the robot):

```
import zmq

from  pypot.server import ZMQRobotServer

robot = ... #create your robot from a config file or using the PoppyHumanoid lib

server = ZMQRobotServer(robot,''0.0.0.0'', 6768)

# We launch the server inside a thread
threading.Thread(target=lambda: server.run()).start()
print ``ready''
```

Client-side code (launch on remote computer):

```
import zmq

c = zmq.Context()
s = c.socket(zmq.REQ)
s.connect (``tcp://poppy.local:6768'') #adapt the hostname or IP to the one of your robot a

#how the read a register
req = {``robot'': {``get_register_value'': {``motor'': ``head_z'', ``register'': ``present_
s.send_json(req)
answer = s.recv_json()
print(answer)

#how to write in a register
req = {``robot'': {``set_register_value'': {``motor'': ``head_z'', ``register'': ``goal_pos
s.send_json(req)
answer = s.recv_json()
print(answer)
```

HTTP method More classical and easy to use. This example uses urllib, but there are other very good Python libraries for HTTP.

To launch the HTTP server on you robot:

```
poppy-services --http
```

the default port is 8080. You can test your connection by directly entering the following URL in your browser: http://poppy.local:8080/motor/list.json

Client-side example code (use on remote computer):

```
import urllib, urllib2, json
```

```
#make a GET request to read the names of all motors
allmotors= urllib2.urlopen(``http://poppy.local:8080/motor/list.json'').read()
print allmotors

#transform json into Python dictionnary
allmotors_dict = json.loads(allmotors)
for m in allmotors_dict[''motors'']:
    print m

#make POST request to move a motor
url = `http://poppy.local:8080/motor/head_z/register/goal_position/value.json'
values = json.dumps(-20)
req = urllib2.Request(url, values)
req.add_header(``Content-Type'','application/json')
response = urllib2.urlopen(req)
```

The logging system

Pypot used the Python's builtin logging module for logging. For details on how to use this module please refer to Python's own documentation or the one on django website. Here, we will only describe what pypot is logging and how it is organised. We will also present a few examples on how to use pypot logging and parse the information.

Logging structure Pypot is logging information at all different levels:

- low-level dynamixel IO
- motor and robot abstraction
- within each primitive
- each request received by the server

  note

  As you probably do not want to log everything (pypot is sending a lot of messages!!!), you have to select in the logging structure what is relevant in your program and define it in your logging configuration.

Pypot's logging naming convention is following pypot's architecture. Here is the detail of what pypot is logging with the associated logger's name:

- The logger's name *pypot.dynamixel.io* is logging information related to opening/closing port (INFO) and each sent/received package (DEBUG). The communication and timeout error are also logged (WARNING). This logger always provides you the port name, the baudrate and timeout of your connection as extra information.
- The logger *pypot.robot.motor* is logging each time a register of a motor is set (DEBUG). The name of the register, the name of the motor and the set value are given in the message.
- *pypot.robot.config* is logging information regarding the creation of a robot through a config dictionary. A message is sent for each motor, controller and alias added (INFO). A WARNING message is also sent when the angle limits of a motor are changed. We provide as extra the entire config dictionary.
- The logger *pypot.robot* is logging when the synchronization is started/stopped (INFO) and when a primitive is attached (INFO).
- *pypot.primitive* logs a message when the primitive is started/stopped and paused/resumed (INFO). Each pypot.primitive.primitive.LoopPrimitive.update of a LoopPrimitive is also logged (DEBUG). Each time a primitive sets a value to a register a message is also logged (DEBUG).

- *pypot.primitive.manager* provides you information on how the values sent within primitives were combined (DEBUG).
- *pypot.server* logs when the server is started (INFO) and each handled request (DEBUG).

Using Pypot's logging

Logging configuration The logging configuration is a dictionnary defining what you want to log: log level, find specific formats as timestamps... More details on Python's logging doc

As an example, let say we want to check the "real" update frequency of a loop primitive. So we specify that we only want the logs coming from 'pypot.primitive' and the message is formatted so we only keep the timestamp:

```
LOGGING = {
    `version': 1,
    `disable_existing_loggers': True,
    `formatters': {
        `time': {
            `format': `%(asctime)s',
        },
    },
    `handlers': {
        `file': {
            `level': `DEBUG',
            `class': `logging.FileHandler',
            `filename': `pypot.log',
            `formatter': `time',
        },
    },
    `loggers': {
        `pypot.primitive': {
            `handlers': ['file', ],
            `level': `DEBUG',
        },
    },
}
```

Now we simply have to set the config to the logging library:

```
logging.config.dictConfig(LOGGING)
```

in your code before starting your primitive and the logs will automatically be collected. At the end of execution, they are saved in a file name *pypot.log*, where each line correspond to a log.

Then if you want, for example, to parse the primitives timestamps logs:

```
t = []

with open(`pypot.log') as f:
    for l in f.readlines():
        d = datetime.datetime.strptime(`%Y-%m-%d %H:%M:%S,%f\n')
        t.append(d)

t = numpy.array(t)
dt = map(lambda dt: dt.total_seconds(), numpy.diff(t))
dt = numpy.array(dt) * 1000
```

```
print(numpy.mean(dt), numpy.std(dt))

plot(dt)
show()
```

Herborist: the configuration tool

Herborist is a graphical tool that helps you detect and configure motors before using them in your robot. See the assembly docs for a tuto.

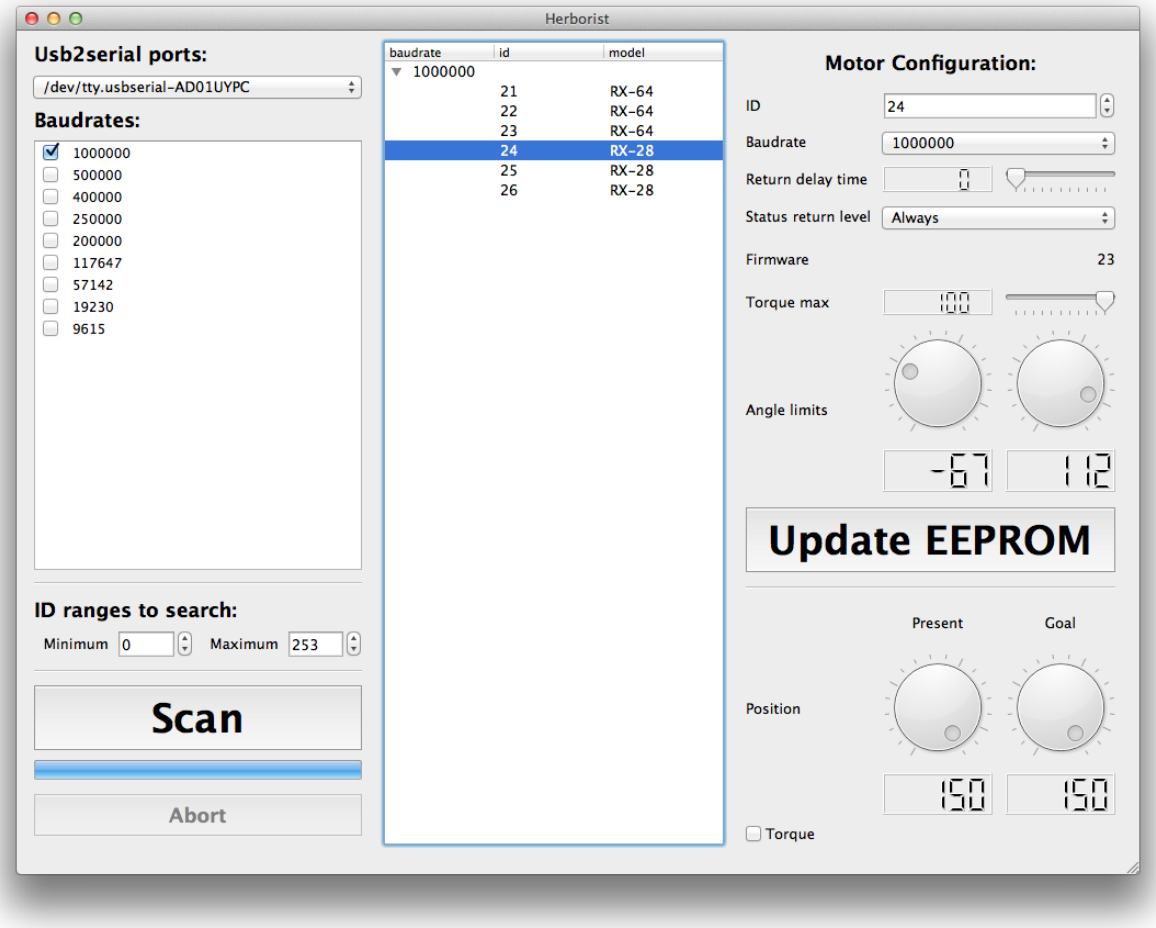
> **warning**
>
> Herborist is entirely written in Python but requires PyQt4 to run.

More precisely, Herborist can be used to:

- Find and identify available serial ports
- Scan multiple baud rates to find all connected motors
- Modify the EEPROM configuration (of single or multiple motors)
- Make motors move (e.g. to test the angle limits).

You can directly launch herborist by running the *herborist* command in your terminal.

> **note**
>
> When you install pypot with the setup.py, herborist is automatically added to your $PATH. You can call it from anywhere thanks to the command:
>
> herborist
>
> You can always find the script in the folder $(PYPOT_SRC)/pypot/tools/herborist.



warning

Herborist is not actively maintained and will hopefully be replaced by a better solution. So its stability is somewhat questionable.

2.2.6 Setup the internal board

Use Poppy ready images

The easiest way to setup the control board of your Poppy Creature is to used one of the pre-made SD-card images. Those images come with everything installed and ready, you just need to copy it on a SD-card and you are good to go. . For that, you will need a free 8Go (or more) SD-card, and download the image corresponding to your board:

- RaspberryPi-2 image for ErgoJr
- Odroid image for Torso or Humanoid

Then to copy it on your SD-card.

Use an RaspberryPi-2 board

All details and resources nessary to make your own "Poppy Ready" image for RaspberryPi-2 can be found on this github repository.

We are here describing the set of tools we use to setup a Raspberry board for a Poppy Creature. While we try to keep this procedure as simple as possible, it still requires a good knowledge of Linux OS and of Python. For those who are not interrested in digging into those details, we will soon provide ready-to-use SD-card images which you can directly download and put in your Raspberry.

While this procedure is mainly designed for the Raspberypi-2, most of it (especially all conda recipes) should also work for the odroid XU4 as both boards used an armv7 CPU. It is important to also note that this procedure is only given as an example: i.e. this is how we build the SD-card images we provide. Yet, you can freely adapt to better match your needs.

Pull Requests are welcomed!

Install Raspbian

Just follow standard instructions from raspberry.org. We use the latest standard Raspbian OS.

Note: the rest of the procedure will require to have an internet access on the Raspberry.

Remove some stuff (~2min)

To save some spaces on the card, we removed some package which are not useful for our needs.

First, make sure the package list is updated:

```
sudo apt-get update
```

Then, remove some package which are not useful here (of course if you need to keep playing minecraft we won't judge you):

```
sudo apt-get remove --purge wolfram-* minecraft-pi scratch penguinspuzzle -y

sudo apt-get autoremove -y
```

Install Python Anaconda (~1min)

To make sure that we can rely on a clean and somewhat better controlled the environment and dependencies we install our own Python environment instead of using the OS packages. If you are not a Python wizard, we strongly advise you to do the same.

We chose the Miniconda distribution has they recently add support for Raspberry and that we can benefit from their recipe concept which makes installation so much faster and simpler.

- Download and install the latest Miniconda

```
wget http://repo.continuum.io/miniconda/Miniconda-latest-Linux-armv7l.sh
bash Miniconda-latest-Linux-armv7l.sh -b
rm Miniconda-latest-Linux-armv7l.sh
```

Create a Poppy Python environment (~1min)

To make things simpler for those who are not familiar with Python ecosystem which can be sometimes confusing, we are creating a dedicated Poppy Environment for the conda distribution (please refer to this doc for details).

```
~/miniconda/bin/conda create -n poppy python=2 -y
source ~/miniconda/bin/activate poppy
```

- Add external poppy-project recipes for conda: we have defined our own recipe and pre-built them so you do not need to compile them on you own board. The full list of available recipes can be seen here (Do not hesitate to contribute and add other recipes!)

```
conda config --add channels poppy-project
conda config --set show_channel_urls True
```

- Set Poppy Environment as default

```
echo ''
export PATH=\$HOME/miniconda/envs/poppy/bin:\$PATH'' >> ~/.bashrc
```

Install the main scientific python packages (~5min)

This is just the basic packages which are used by most poppy softwares and demos:

```
conda install jupyter numpy scipy matplotlib -y
```

Poppy Creature Packages (~1min)

Each Poppy Creature comes with its own specific software. Each of this package is based on pypot which handles all the low-level communication with the robot.

So, first we install pypot:

```
conda install pypot -y
```

Then, the software for your specific creature. For isntance, if you want to use an Ergo-Jr:

```
conda install poppy-ergo-jr -y
```

The list of available poppy creatures can be found via:

```
conda search poppy
```

Note that you can install as many creatures as you want.

Camera support (~1min) - *Raspberry only!*

First you need to setup your camera according to the official documentation. You do not need to install the picamera package as we will not use it but uses opencv instead.

Once you have enabled the camera using raspi-config, you can check if it worked via:

```
raspistill -o cam.jpg
```

*Note this will only work if you boot to the Desktop!**

Then, we launch the driver:

```
sudo modprobe bcm2835-v4l2
```

You can also configure the OS so it automatically launches the driver at startup (otherwise you have to use the previous line each time you reboot your card):

```
sudo sed -i /bcm2835-v4l2/d /etc/modules
echo ``bcm2835-v4l2'' | sudo tee -a /etc/modules
```

OpenCV and Image features Then we install the standard computer vision library OpenCV using a conda recipe as well as it is very long to compile. Make sure it will install the version 3.

```
conda install opencv -y
```

For marker detection we use the hampy library:

```
conda install hampy -y
```

Troubleshooting

Please check and report any unknown issue!

Use an ODROID board

2.2.7 APIs

poppy_humanoid package

Subpackages

poppy_humanoid.primitives package

Submodules

poppy_humanoid.primitives.dance module

poppy_humanoid.primitives.idle module

poppy_humanoid.primitives.interaction module

poppy_humanoid.primitives.posture module

poppy_humanoid.primitives.safe module

Module contents

Submodules

poppy_humanoid.poppy_humanoid module

Module contents

poppy_torso package

Subpackages

poppy_torso.primitives package

Submodules

poppy_torso.primitives.dance module

poppy_torso.primitives.idle module

poppy_torso.primitives.interaction module

poppy_torso.primitives.posture module

poppy_torso.primitives.safe module

Module contents

poppy_torso.utils package

Submodules

poppy_torso.utils.min_jerk module

Module contents

Submodules

poppy_torso.poppy_torso module

Module contents

poppy_ergo_jr package

Submodules

poppy_ergo_jr.dance module

poppy_ergo_jr.headfollow module

poppy_ergo_jr.jump module

poppy_ergo_jr.poppy_ergo_jr module

poppy_ergo_jr.postures module

Module contents

poppy-creature package

Subpackages

poppy.creatures package

Submodules

poppy.creatures.abstractcreature module

poppy.creatures.poppy_sim module

poppy.creatures.services_launcher module

poppy.creatures.snap_launcher module

Module contents

Module contents

pypot package

Subpackages

pypot.dynamixel package

Subpackages

pypot.dynamixel.io package

Submodules

pypot.dynamixel.io.abstract_io module

class pypot.dynamixel.io.abstract_io.**AbstractDxlIO**(*port*, *baudrate=1000000*, *timeout=0.05*, *use_sync_read=False*, *error_handler_cls=None*, *convert=True*)

Bases: pypot.robot.io.AbstractIO

Low-level class to handle the serial communication with the robotis motors.

At instanciation, it opens the serial port and sets the communication parameters.

Parameters

- **port** (*string*) – the serial port to use (e.g. Unix (/dev/tty...), Windows (COM...)).
- **baudrate** (*int*) – default for new motors: 57600, for PyPot motors: 1000000
- **timeout** (*float*) – read timeout in seconds
- **use_sync_read** (*bool*) – whether or not to use the SYNC_READ instruction
- **error_handler** (*DxlErrorHandler*) – set a handler that will receive the different errors
- **convert** (*bool*) – whether or not convert values to units expressed in the standard system

Raises DxlError if the port is already used.

classmethod get_used_ports()

open(*port*, *baudrate=1000000*, *timeout=0.05*)

Opens a new serial communication (closes the previous communication if needed).

Raises DxlError if the port is already used.

close(*_force_lock=False*)

Closes the serial communication if opened.

flush(*_force_lock=False*)

Flushes the serial communication (both input and output).

port

Port used by the DxlIO. If set, will re-open a new connection.

baudrate

Baudrate used by the DxlIO. If set, will re-open a new connection.

timeout

Timeout used by the DxlIO. If set, will re-open a new connection.

closed

Checks if the connection is closed.

ping(*id*)
Pings the motor with the specified id.

Note: The motor id should always be included in [0, 253]. 254 is used for broadcast.

scan(*ids=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253]*)
Pings all ids within the specified list, by default it finds all the motors connected to the bus.

get_model(*ids*)
Gets the model for the specified motors.

change_id(*new_id_for_id*)
Changes the id of the specified motors (each id must be unique on the bus).

change_baudrate(*baudrate_for_ids*)
Changes the baudrate of the specified motors.

get_status_return_level(*ids*, ***kwargs*)
Gets the status level for the specified motors.

set_status_return_level(*srl_for_id*, ***kwargs*)
Sets status return level to the specified motors.

switch_led_on(*ids*)
Switches on the LED of the motors with the specified ids.

switch_led_off(*ids*)
Switches off the LED of the motors with the specified ids.

enable_torque(*ids*)
Enables torque of the motors with the specified ids.

disable_torque(*ids*)
Disables torque of the motors with the specified ids.

get_pid_gain(*ids*, ***kwargs*)
Gets the pid gain for the specified motors.

set_pid_gain(*pid_for_id*, ***kwargs*)
Sets the pid gain to the specified motors.

get_control_table(*ids*, ***kwargs*)
Gets the full control table for the specified motors.

..note:: This function requires the model for each motor to be known. Querring this additional information might add some extra delay.

exception pypot.dynamixel.io.abstract_io.**DxlError**
Bases: exceptions.Exception

Base class for all errors encountered using DxlIO.

exception pypot.dynamixel.io.abstract_io.**DxlCommunicationError**(*dxl_io*, *message*, *instruction_packet*)

Bases: *pypot.dynamixel.io.abstract_io.DxlError*

Base error for communication error encountered when using DxlIO.

exception pypot.dynamixel.io.abstract_io.**DxlTimeoutError**(*dxl_io*, *instruction_packet*, *ids*)

Bases: *pypot.dynamixel.io.abstract_io.DxlCommunicationError*

Timeout error encountered when using DxlIO.

pypot.dynamixel.io.abstract_io.**with_True**(**args*, ***kwds*)

pypot.dynamixel.io.io module

class pypot.dynamixel.io.io.**DxlIO**(*port*, *baudrate=1000000*, *timeout=0.05*, *use_sync_read=False*, *error_handler_cls=None*, *convert=True*)

Bases: *pypot.dynamixel.io.abstract_io.AbstractDxlIO*

At instanciation, it opens the serial port and sets the communication parameters.

Parameters

- **port** (*string*) – the serial port to use (e.g. Unix (/dev/tty...), Windows (COM...)).
- **baudrate** (*int*) – default for new motors: 57600, for PyPot motors: 1000000
- **timeout** (*float*) – read timeout in seconds
- **use_sync_read** (*bool*) – whether or not to use the SYNC_READ instruction
- **error_handler** (*DxlErrorHandler*) – set a handler that will receive the different errors
- **convert** (*bool*) – whether or not convert values to units expressed in the standard system

Raises DxlError if the port is already used.

factory_reset()

Reset all motors on the bus to their factory default settings.

get_control_mode(*ids*)

Gets the mode ('joint' or 'wheel') for the specified motors.

set_wheel_mode(*ids*)

Sets the specified motors to wheel mode.

set_joint_mode(*ids*)

Sets the specified motors to joint mode.

set_control_mode(*mode_for_id*)

set_angle_limit(*limit_for_id*, ***kwargs*)

Sets the angle limit to the specified motors.

get_alarm_LED(*ids*, ***kwargs*)

Gets alarm LED from the specified motors.

get_alarm_shutdown(*ids*, ***kwargs*)

Gets alarm shutdown from the specified motors.

get_angle_limit(*ids*, ***kwargs*)

Gets angle limit from the specified motors.

get_compliance_margin(*ids*, ***kwargs*)
Gets compliance margin from the specified motors.

get_compliance_slope(*ids*, ***kwargs*)
Gets compliance slope from the specified motors.

get_drive_mode(*ids*, ***kwargs*)
Gets drive mode from the specified motors.

get_firmware(*ids*, ***kwargs*)
Gets firmware from the specified motors.

get_goal_position(*ids*, ***kwargs*)
Gets goal position from the specified motors.

get_goal_position_speed_load(*ids*, ***kwargs*)
Gets goal position speed load from the specified motors.

get_highest_temperature_limit(*ids*, ***kwargs*)
Gets highest temperature limit from the specified motors.

get_max_torque(*ids*, ***kwargs*)
Gets max torque from the specified motors.

get_moving_speed(*ids*, ***kwargs*)
Gets moving speed from the specified motors.

get_present_load(*ids*, ***kwargs*)
Gets present load from the specified motors.

get_present_position(*ids*, ***kwargs*)
Gets present position from the specified motors.

get_present_position_speed_load(*ids*, ***kwargs*)
Gets present position speed load from the specified motors.

get_present_speed(*ids*, ***kwargs*)
Gets present speed from the specified motors.

get_present_temperature(*ids*, ***kwargs*)
Gets present temperature from the specified motors.

get_present_voltage(*ids*, ***kwargs*)
Gets present voltage from the specified motors.

get_return_delay_time(*ids*, ***kwargs*)
Gets return delay time from the specified motors.

get_torque_limit(*ids*, ***kwargs*)
Gets torque limit from the specified motors.

get_voltage_limit(*ids*, ***kwargs*)
Gets voltage limit from the specified motors.

is_led_on(*ids*, ***kwargs*)
Gets LED from the specified motors.

is_moving(*ids*, ***kwargs*)
Gets moving from the specified motors.

is_torque_enabled(*ids*, ***kwargs*)
Gets torque_enable from the specified motors.

set_alarm_LED(*value_for_id*, ***kwargs*)
Sets alarm LED to the specified motors.

set_alarm_shutdown(*value_for_id*, ***kwargs*)
Sets alarm shutdown to the specified motors.

set_compliance_margin(*value_for_id*, ***kwargs*)
Sets compliance margin to the specified motors.

set_compliance_slope(*value_for_id*, ***kwargs*)
Sets compliance slope to the specified motors.

set_drive_mode(*value_for_id*, ***kwargs*)
Sets drive mode to the specified motors.

set_goal_position(*value_for_id*, ***kwargs*)
Sets goal position to the specified motors.

set_goal_position_speed_load(*value_for_id*, ***kwargs*)
Sets goal position speed load to the specified motors.

set_highest_temperature_limit(*value_for_id*, ***kwargs*)
Sets highest temperature limit to the specified motors.

set_max_torque(*value_for_id*, ***kwargs*)
Sets max torque to the specified motors.

set_moving_speed(*value_for_id*, ***kwargs*)
Sets moving speed to the specified motors.

set_return_delay_time(*value_for_id*, ***kwargs*)
Sets return delay time to the specified motors.

set_torque_limit(*value_for_id*, ***kwargs*)
Sets torque limit to the specified motors.

set_voltage_limit(*value_for_id*, ***kwargs*)
Sets voltage limit to the specified motors.

pypot.dynamixel.io.io_320 module

class pypot.dynamixel.io.io_320.**Dxl320IO**(*port*, *baudrate=1000000*, *timeout=0.05*, *use_sync_read=False*, *error_handler_cls=None*, *convert=True*)

Bases: *pypot.dynamixel.io.abstract_io.AbstractDxlIO*

At instanciation, it opens the serial port and sets the communication parameters.

Parameters

- **port** (*string*) – the serial port to use (e.g. Unix (/dev/tty...), Windows (COM...)).
- **baudrate** (*int*) – default for new motors: 57600, for PyPot motors: 1000000
- **timeout** (*float*) – read timeout in seconds
- **use_sync_read** (*bool*) – whether or not to use the SYNC_READ instruction
- **error_handler** (*DxlErrorHandler*) – set a handler that will receive the different errors
- **convert** (*bool*) – whether or not convert values to units expressed in the standard system

Raises DxlError if the port is already used.

set_wheel_mode(*ids*)

set_joint_mode(*ids*)

get_goal_position_speed_load(*ids*)

set_goal_position_speed_load(*value_for_ids*)

factory_reset(*ids*, *except_ids=False*, *except_baudrate_and_ids=False*)
Reset all motors on the bus to their factory default settings.

get_LED_color(*ids*, ***kwargs*)
Gets LED color from the specified motors.

get_alarm_shutdown(*ids*, ***kwargs*)
Gets alarm shutdown from the specified motors.

get_angle_limit(*ids*, ***kwargs*)
Gets angle limit from the specified motors.

get_control_mode(*ids*, ***kwargs*)
Gets control mode from the specified motors.

get_firmware(*ids*, ***kwargs*)
Gets firmware from the specified motors.

get_goal_position(*ids*, ***kwargs*)
Gets goal position from the specified motors.

get_highest_temperature_limit(*ids*, ***kwargs*)
Gets highest temperature limit from the specified motors.

get_max_torque(*ids*, ***kwargs*)
Gets max torque from the specified motors.

get_moving_speed(*ids*, ***kwargs*)
Gets moving speed from the specified motors.

get_present_load(*ids*, ***kwargs*)
Gets present load from the specified motors.

get_present_position(*ids*, ***kwargs*)
Gets present position from the specified motors.

get_present_position_speed_load(*ids*, ***kwargs*)
Gets present position speed load from the specified motors.

get_present_speed(*ids*, ***kwargs*)
Gets present speed from the specified motors.

get_present_temperature(*ids*, ***kwargs*)
Gets present temperature from the specified motors.

get_present_voltage(*ids*, ***kwargs*)
Gets present voltage from the specified motors.

get_return_delay_time(*ids*, ***kwargs*)
Gets return delay time from the specified motors.

get_torque_limit(*ids*, ***kwargs*)
Gets torque limit from the specified motors.

get_voltage_limit(*ids*, ***kwargs*)
Gets voltage limit from the specified motors.

is_led_on(*ids*, ***kwargs*)
Gets LED from the specified motors.

is_moving(*ids*, ***kwargs*)
Gets moving from the specified motors.

is_torque_enabled(*ids*, ***kwargs*)
Gets torque_enable from the specified motors.

set_LED_color(*value_for_id*, ***kwargs*)
Sets LED color to the specified motors.

set_alarm_shutdown(*value_for_id*, ***kwargs*)
Sets alarm shutdown to the specified motors.

set_angle_limit(*value_for_id*, ***kwargs*)
Sets angle limit to the specified motors.

set_control_mode(*value_for_id*, ***kwargs*)
Sets control mode to the specified motors.

set_goal_position(*value_for_id*, ***kwargs*)
Sets goal position to the specified motors.

set_highest_temperature_limit(*value_for_id*, ***kwargs*)
Sets highest temperature limit to the specified motors.

set_max_torque(*value_for_id*, ***kwargs*)
Sets max torque to the specified motors.

set_moving_speed(*value_for_id*, ***kwargs*)
Sets moving speed to the specified motors.

set_return_delay_time(*value_for_id*, ***kwargs*)
Sets return delay time to the specified motors.

set_torque_limit(*value_for_id*, ***kwargs*)
Sets torque limit to the specified motors.

set_voltage_limit(*value_for_id*, ***kwargs*)
Sets voltage limit to the specified motors.

Module contents

pypot.dynamixel.protocol package

Submodules

pypot.dynamixel.protocol.v1 module

class pypot.dynamixel.protocol.v1.**DxlInstruction**
Bases: object

PING = 1

READ_DATA = 2

WRITE_DATA = 3

RESET = 6

SYNC_WRITE = 131

SYNC_READ = 132

class pypot.dynamixel.protocol.v1.**DxlPacketHeader**
Bases: pypot.dynamixel.protocol.v1.DxlPacketHeader

This class represents the header of a Dxl Packet.

They are constructed as follows [0xFF, 0xFF, ID, LENGTH] where:

- ID represents the ID of the motor who received (resp. sent) the intruction (resp. status) packet.
- LENGTH represents the length of the rest of the packet

length = 4

marker = bytearray(b'\xff\xff')

classmethod from_string(*data*)

class pypot.dynamixel.protocol.v1.**DxlInstructionPacket**
Bases: pypot.dynamixel.protocol.v1.DxlInstructionPacket

This class is used to represent a dynamixel instruction packet.

An instruction packet is constructed as follows: [0xFF, 0xFF, ID, LENGTH, INSTRUCTION, PARAM 1, PARAM 2, ..., PARAM N, CHECKSUM]

(for more details see http://support.robotis.com/en/product/dxl_main.htm)

to_array()

to_string()

length

checksum

class pypot.dynamixel.protocol.v1.**DxlPingPacket**
Bases: pypot.dynamixel.protocol.v1.DxlInstructionPacket

This class is used to represent ping packet.

class pypot.dynamixel.protocol.v1.**DxlResetPacket**
Bases: pypot.dynamixel.protocol.v1.DxlInstructionPacket

This class is used to represent reset packet.

class pypot.dynamixel.protocol.v1.**DxlReadDataPacket**
Bases: pypot.dynamixel.protocol.v1.DxlInstructionPacket

This class is used to represent read data packet (to read value).

class pypot.dynamixel.protocol.v1.**DxlSyncReadPacket**
Bases: pypot.dynamixel.protocol.v1.DxlInstructionPacket

This class is used to represent sync read packet (to synchronously read values).

class pypot.dynamixel.protocol.v1.**DxlWriteDataPacket**
Bases: pypot.dynamixel.protocol.v1.DxlInstructionPacket

This class is used to reprensent write data packet (to write value).

class pypot.dynamixel.protocol.v1.**DxlSyncWritePacket**
Bases: pypot.dynamixel.protocol.v1.DxlInstructionPacket

This class is used to represent sync write packet (to synchronously write values).

class pypot.dynamixel.protocol.v1.**DxlStatusPacket**
Bases: pypot.dynamixel.protocol.v1.DxlStatusPacket

This class is used to represent a dynamixel status packet.

A status packet is constructed as follows: [0xFF, 0xFF, ID, LENGTH, ERROR, PARAM 1, PARAM 2, ..., PARAM N, CHECKSUM]

(for more details see http://support.robotis.com/en/product/dxl_main.htm)

classmethod **from_string**(*data*)

pypot.dynamixel.protocol.v2 module

class pypot.dynamixel.protocol.v2.**DxlInstruction**

Bases: object

PING = 1

READ_DATA = 2

WRITE_DATA = 3

RESET = 6

SYNC_READ = 130

SYNC_WRITE = 131

class pypot.dynamixel.protocol.v2.**DxlPacketHeader**

Bases: pypot.dynamixel.protocol.v2.DxlPacketHeader

This class represents the header of a Dxl Packet.

They are constructed as follows [0xFF, 0xFF, 0xFD, 0x00, ID, LEN_L, LEN_H] where:

- ID represents the ID of the motor who received (resp. sent) the intruction (resp. status) packet.
- LEN_L, LEN_H represents the length of the rest of the packet

length = 7

marker = bytearray(b'\xff\xff\xfd\x00')

classmethod **from_string**(*data*)

class pypot.dynamixel.protocol.v2.**DxlInstructionPacket**

Bases: pypot.dynamixel.protocol.v2.DxlInstructionPacket

This class is used to represent a dynamixel instruction packet.

An instruction packet is constructed as follows: [0xFF, 0xFF, 0xFD, 0x00, ID, LEN_L, LEN_H, INST, PARAM 1, PARAM 2, ..., PARAM N, CRC_L, CRC_H]

(for more details see http://support.robotis.com/en/product/dxl_main.htm)

to_array()

to_string()

length

checksum

class pypot.dynamixel.protocol.v2.**DxlPingPacket**

Bases: pypot.dynamixel.protocol.v2.DxlInstructionPacket

This class is used to represent ping packet.

class pypot.dynamixel.protocol.v2.**DxlResetPacket**
Bases: pypot.dynamixel.protocol.v2.DxlInstructionPacket

This class is used to represent factory reset packet.

class pypot.dynamixel.protocol.v2.**DxlReadDataPacket**
Bases: pypot.dynamixel.protocol.v2.DxlInstructionPacket

This class is used to represent read data packet (to read value).

class pypot.dynamixel.protocol.v2.**DxlSyncReadPacket**
Bases: pypot.dynamixel.protocol.v2.DxlInstructionPacket

This class is used to represent sync read packet (to synchronously read values).

class pypot.dynamixel.protocol.v2.**DxlWriteDataPacket**
Bases: pypot.dynamixel.protocol.v2.DxlInstructionPacket

This class is used to reprensent write data packet (to write value).

class pypot.dynamixel.protocol.v2.**DxlSyncWritePacket**
Bases: pypot.dynamixel.protocol.v2.DxlInstructionPacket

This class is used to represent sync write packet (to synchronously write values).

class pypot.dynamixel.protocol.v2.**DxlStatusPacket**
Bases: pypot.dynamixel.protocol.v2.DxlStatusPacket

This class is used to represent a dynamixel status packet.

A status packet is constructed as follows: [0xFF, 0xFF, 0xFD, 0x00, ID, LEN_L, LEN_H, 0x55, ERROR, PARAM 1, PARAM 2, ..., PARAM N, CRC_L, CRC_H]

(for more details see http://support.robotis.com/en/product/dxl_main.htm)

classmethod from_string(*data*)

pypot.dynamixel.protocol.v2.**crc16**(*data_blk*, *data_blk_size*, *crc_accum=0*)

Module contents

Submodules

pypot.dynamixel.controller module

class pypot.dynamixel.controller.**DxlController**(*io*, *motors*, *sync_freq*, *synchronous*, *mode*, *regname*, *varname=None*)
Bases: pypot.robot.controller.MotorsController

working_motors

synced_motors

setup()

update()

get_register(*motors*)
Gets the value from the specified register and sets it to the *DxlMotor*.

set_register(*motors*)
Gets the value from *DxlMotor* and sets it to the specified register.

class pypot.dynamixel.controller.**AngleLimitRegisterController**(*io*, *motors*, *sync_freq*, *synchronous*)

Bases: *pypot.dynamixel.controller.DxlController*

synced_motors

get_register(*motors*)

class pypot.dynamixel.controller.**PosSpeedLoadDxlController**(*io*, *motors*, *sync_freq*)

Bases: *pypot.dynamixel.controller.DxlController*

setup()

update()

get_present_position_speed_load(*motors*)

set_goal_position_speed_load(*motors*)

pypot.dynamixel.conversion module This module describes all the conversion method used to transform value from the representation used by the dynamixel motor to a more standard form (e.g. degrees, volt...).

For compatibility issue all comparison method should be written in the following form (even if the model is not actually used):

- def my_conversion_from_dxl_to_si(value, model): ...
- def my_conversion_from_si_to_dxl(value, model): ...

Note: If the control is readonly you only need to write the dxl_to_si conversion.

pypot.dynamixel.conversion.**dxl_to_degree**(*value*, *model*)

pypot.dynamixel.conversion.**degree_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_speed**(*value*, *model*)

pypot.dynamixel.conversion.**speed_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_torque**(*value*, *model*)

pypot.dynamixel.conversion.**torque_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_load**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_pid**(*value*, *model*)

pypot.dynamixel.conversion.**pid_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_model**(*value*, *dummy=None*)

pypot.dynamixel.conversion.**check_bit**(*value*, *offset*)

pypot.dynamixel.conversion.**dxl_to_drive_mode**(*value*, *model*)

pypot.dynamixel.conversion.**drive_mode_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_baudrate**(*value*, *model*)

pypot.dynamixel.conversion.**baudrate_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_rdt**(*value*, *model*)

pypot.dynamixel.conversion.**rdt_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_temperature**(*value*, *model*)

pypot.dynamixel.conversion.**temperature_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_voltage**(*value*, *model*)

pypot.dynamixel.conversion.**voltage_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_status**(*value*, *model*)

pypot.dynamixel.conversion.**status_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_alarm**(*value*, *model*)

pypot.dynamixel.conversion.**decode_error**(*error_code*)

pypot.dynamixel.conversion.**alarm_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**XL320LEDColors**
alias of Colors

pypot.dynamixel.conversion.**dxl_to_led_color**(*value*, *model*)

pypot.dynamixel.conversion.**led_color_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_control_mode**(*value*, _)

pypot.dynamixel.conversion.**control_mode_to_dxl**(*mode*, _)

pypot.dynamixel.conversion.**dxl_to_bool**(*value*, *model*)

pypot.dynamixel.conversion.**bool_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_decode**(*data*)

pypot.dynamixel.conversion.**dxl_decode_all**(*data*, *nb_elem*)

pypot.dynamixel.conversion.**dxl_code**(*value*, *length*)

pypot.dynamixel.conversion.**dxl_code_all**(*value*, *length*, *nb_elem*)

pypot.dynamixel.error module

class pypot.dynamixel.error.**DxlErrorHandler**

Bases: object

This class is used to represent all the error that you can/should handle.

The errors can be of two types:

- communication error (timeout, communication)
- motor error (voltage, limit, overload...)

This class was designed as an abstract class and so you should write your own handler by subclassing this class and defining the apropriate behavior for your program.

> **Warning:** The motor error should be overload carrefuly as they can indicate important mechanical issue.

handle_timeout(*timeout_error*)

handle_communication_error(*communication_error*)

handle_input_voltage_error(*instruction_packet*)

handle_angle_limit_error(*instruction_packet*)

handle_overheating_error(*instruction_packet*)

handle_range_error(*instruction_packet*)

handle_checksum_error(*instruction_packet*)

handle_overload_error(*instruction_packet*)

handle_instruction_error(*instruction_packet*)

handle_none_error(*instruction_packet*)

class pypot.dynamixel.error.**BaseErrorHandler**
Bases: *pypot.dynamixel.error.DxlErrorHandler*

This class is a basic handler that just skip the communication errors.

handle_timeout(*timeout_error*)

handle_communication_error(*com_error*)

handle_none_error(*instruction_packet*)

pypot.dynamixel.motor module

class pypot.dynamixel.motor.**DxlRegister**(*rw=False*)
Bases: object

class pypot.dynamixel.motor.**DxlOrientedRegister**(*rw=False*)
Bases: *pypot.dynamixel.motor.DxlRegister*

class pypot.dynamixel.motor.**DxlPositionRegister**(*rw=False*)
Bases: *pypot.dynamixel.motor.DxlOrientedRegister*

class pypot.dynamixel.motor.**RegisterOwner**
Bases: type

class pypot.dynamixel.motor.**DxlMotor**(*id*, *name=None*, *model=''*, *direct=True*, *offset=0.0*, *broken=False*)
Bases: pypot.robot.motor.Motor

High-level class used to represent and control a generic dynamixel motor.

This class provides all level access to (see *registers* for an exhaustive list):

- motor id
- motor name
- motor model
- present position/speed/load
- goal position/speed/load
- compliant
- motor orientation and offset
- angle limit
- temperature
- voltage

This class represents a generic robotis motor and you define your own subclass for specific motors (see *DxlMXMotor* or *DxlAXRXMotor*).

Those properties are synchronized with the real motors values thanks to a *DxlController*.

registers = ['registers', 'goal_speed', 'compliant', 'safe_compliant', 'angle_limit', 'present_load', 'id', 'present_tempe

goal_speed
Goal speed (in degrees per second) of the motor.

This property can be used to control your motor in speed. Setting a goal speed will automatically change the moving speed and sets the goal position as the angle limit.

Note: The motor will turn until reaching the angle limit. But this is not a wheel mode, so the motor will stop at its limits.

---

compliant_behavior

compliant

angle_limit

goto_behavior

goto_position(*position*, *duration*, *control=None*, *wait=False*)
Automatically sets the goal position and the moving speed to reach the desired position within the duration.

class pypot.dynamixel.motor.**DxlAXRXMotor**(*id*, *name=None*, *model=''*, *direct=True*, *offset=0.0*, *broken=False*)
Bases: *pypot.dynamixel.motor.DxlMotor*

This class represents the AX robotis motor.

This class adds access to:

- compliance margin/slope (see the robotis website for details)

registers = ['registers', 'goal_speed', 'compliant', 'safe_compliant', 'angle_limit', 'present_load', 'id', 'present_tempe

class pypot.dynamixel.motor.**DxlMXMotor**(*id*, *name=None*, *model=''*, *direct=True*, *offset=0.0*, *broken=False*)
Bases: *pypot.dynamixel.motor.DxlMotor*

This class represents the RX and MX robotis motor.

This class adds access to:

- PID gains (see the robotis website for details)

This class represents the RX and MX robotis motor.

This class adds access to:

- PID gains (see the robotis website for details)

registers = ['registers', 'goal_speed', 'compliant', 'safe_compliant', 'angle_limit', 'present_load', 'id', 'present_tempe

class pypot.dynamixel.motor.**DxlXL320Motor**(*id*, *name=None*, *model='XL-320'*, *direct=True*, *offset=0.0*, *broken=False*)
Bases: *pypot.dynamixel.motor.DxlMXMotor*

registers = ['registers', 'goal_speed', 'compliant', 'safe_compliant', 'angle_limit', 'present_load', 'id', 'present_tempe

class pypot.dynamixel.motor.**SafeCompliance**(*motor*, *frequency=50*)
Bases: pypot.utils.stoppablethread.StoppableLoopThread

This class creates a controller to active compliance only if the current motor position is included in the angle limit, else the compliance is turned off.

update()

teardown()

pypot.dynamixel.syncloop module

class pypot.dynamixel.syncloop.**MetaDxlController**(*io*, *motors*, *controllers*)

Bases: pypot.robot.controller.MotorsController

Synchronizes the reading/writing of *DxlMotor* with the real motors.

This class handles synchronization loops that automatically read/write values from the "software" *DxlMotor* with their "hardware" equivalent. Those loops shared a same DxlIO connection to avoid collision in the bus. Each loop run within its own thread as its own frequency.

> **Warning:** As all the loop attached to a controller shared the same bus, you should make sure that they can run without slowing down the other ones.

setup()

Starts all the synchronization loops.

update()

teardown()

Stops the synchronization loops.

class pypot.dynamixel.syncloop.**BaseDxlController**(*io*, *motors*)

Bases: *pypot.dynamixel.syncloop.MetaDxlController*

Implements a basic controller that synchronized the most frequently used values.

More precisely, this controller:

- reads the present position, speed, load at 50Hz
- writes the goal position, moving speed and torque limit at 50Hz
- writes the pid gains (or compliance margin and slope) at 10Hz
- reads the present voltage and temperature at 1Hz

class pypot.dynamixel.syncloop.**LightDxlController**(*io*, *motors*)

Bases: *pypot.dynamixel.syncloop.MetaDxlController*

Module contents

pypot.dynamixel.**get_available_ports**(*only_free=False*)

pypot.dynamixel.**get_port_vendor_info**(*port=None*)

Return vendor informations of a usb2serial device. It may depends on the Operating System. :param string port: port of the usb2serial device

Example

Result with a USB2Dynamixel on Linux: In [1]: import pypot.dynamixel In [2]: pypot.dynamixel.get_port_vendor_info('/dev/ttyUSB0') Out[2]: 'USB VID:PID=0403:6001 SNR=A7005LKE'

pypot.dynamixel.**find_port**(*ids*, *strict=True*)

Find the port with the specified attached motor ids.

Parameters

- **ids** (*list*) – list of motor ids to find
- **strict** (*bool*) – specify if all ids should be find (when set to False, only half motor must be found)

> **Warning:** If two (or more) ports are attached to the same list of motor ids the first match will be returned.

pypot.dynamixel.**autodetect_robot**()
Creates a Robot by detecting dynamixel motors on all available ports.

pypot.primitive package

Submodules

pypot.primitive.manager module

class pypot.primitive.manager.**PrimitiveManager**(*motors*, *freq=50*, *filter=<functools.partial object>*)
Bases: pypot.utils.stoppablethread.StoppableLoopThread

Combines all Primitive orders and affect them to the real motors.

At a predefined frequency, the manager gathers all the orders sent by the primitive to the "fake" motors, combined them thanks to the filter function and affect them to the "real" motors.

Note: The primitives are automatically added (resp. removed) to the manager when they are started (resp. stopped).

Parameters

- **motors** (list of *DxlMotor*) – list of real motors used by the attached primitives
- **freq** (*int*) – update frequency
- **filter** (*func*) – function used to combine the different request (default mean)

add(*p*)
Add a primitive to the manager. The primitive automatically attached itself when started.

remove(*p*)
Remove a primitive from the manager. The primitive automatically remove itself when stopped.

primitives
List of all attached Primitive.

update()
Combined at a predefined frequency the request orders and affect them to the real motors.

stop()
Stop the primitive manager.

pypot.primitive.move module

class pypot.primitive.move.**Move**(*freq*)
Bases: object

Simple class used to represent a movement.

This class simply wraps a sequence of positions of specified motors. The sequence must be recorded at a predefined frequency. This move can be recorded through the MoveRecorder class and played thanks to a MovePlayer.

framerate

add_position(*pos*, *time*)
Add a new position to the movement sequence.

Each position is typically stored as a dict of (time, (motor_name,motor_position)).

iterpositions()
Returns an iterator on the stored positions.

positions()
Returns a copy of the stored positions.

save(*file*)
Saves the `Move` to a json file.

Note: The format used to store the `Move` is extremely verbose and should be obviously optimized for long moves.

---

classmethod load(*file*)
Loads a `Move` from a json file.

class pypot.primitive.move.**MoveRecorder**(*robot*, *freq*, *tracked_motors*)
Bases: pypot.primitive.primitive.LoopPrimitive

Primitive used to record a `Move`.

The recording can be `start()` and `stop()` by using the `LoopPrimitive` methods.

Note: Re-starting the recording will create a new `Move` losing all the previously stored data.

---

setup()

update()

move
Returns the currently recorded `Move`.

add_tracked_motors(*tracked_motors*)
Add new motors to the recording

class pypot.primitive.move.**MovePlayer**(*robot*, *move=None*, *play_speed=1.0*, *move_filename=None*, *start_max_speed=50*, ***kwargs*)
Bases: pypot.primitive.primitive.LoopPrimitive

Primitive used to play a `Move`.

The playing can be `start()` and `stop()` by using the `LoopPrimitive` methods.

Warning: the primitive is run automatically the same framerate than the move record. The play_speed attribute change only time lockup/interpolation

setup()

update()

duration()

pypot.primitive.primitive module

class pypot.primitive.primitive.**Primitive**(*robot*)
Bases: pypot.utils.stoppablethread.StoppableThread

A Primitive is an elementary behavior that can easily be combined to create more complex behaviors.

A primitive is basically a thread with access to a "fake" robot to ensure a sort of sandboxing. More precisely, it means that the primitives will be able to:

- request values from the real robot (motor values, sensors or attached primitives)
- request modification of motor values (those calls will automatically be combined among all primitives by the `PrimitiveManager`).

The syntax of those requests directly match the equivalent code that you could write from the `Robot`. For instance you can write:

```
class MyPrimitive(Primitive):
    def run(self):
        while True:
            for m in self.robot.motors:
                m.goal_position = m.present_position + 10

        time.sleep(1)
```

Warning: In the example above, while it seems that you are setting a new goal_position, you are only requesting it. In particular, another primitive could request another goal_position and the result will be the combination of both request. For example, if you have two primitives: one setting the goal_position to 10 and the other setting the goal_position to -20, the real goal_position will be set to -5 (by default the mean of all request is used, see the `PrimitiveManager` class for details).

Primitives were developed to allow for the creation of complex behaviors such as walking. You could imagine - and this is what is actually done on the Poppy robot - having one primitive for the walking gait, another for the balance and another for handling falls.

Note: This class should always be extended to define your particular behavior in the `run()` method.

At instanciation, it automatically transforms the `Robot` into a `MockupRobot`.

Warning: You should not directly pass motors as argument to the primitive. If you need to, use the method `get_mockup_motor()` to transform them into "fake" motors. See the write_own_prim section for details.

methods = ['start', 'stop', 'pause', 'resume']

properties = []

setup()
Setup methods called before the run loop.

You can override this method to setup the environment needed by your primitive before the run loop. This method will be called every time the primitive is started/restarted.

run()
Run method of the primitive thread. You should always overwrite this method.

Warning: You are responsible of handling the `should_stop()`, `should_pause()` and `wait_to_resume()` methods correctly so the code inside your run function matches the desired behavior. You can refer to the code of the `run()` method of the `LoopPrimitive` as an example.

After termination of the run function, the primitive will automatically be removed from the list of active primitives of the `PrimitiveManager`.

teardown()
Tear down methods called after the run loop.

You can override this method to clean up the environment needed by your primitive. This method will be called every time the primitive is stopped.

elapsed_time
Elapsed time (in seconds) since the primitive runs.

start()
Start or restart (the `stop()` method will automatically be called) the primitive.

stop(*wait=True*)
Requests the primitive to stop.

is_alive()
Determines whether the primitive is running or not.

The value will be true only when the `run()` function is executed.

get_mockup_motor(*motor*)
Gets the equivalent `MockupMotor`.

class pypot.primitive.primitive.**LoopPrimitive**(*robot*, *freq*)
Bases: pypot.primitive.primitive.Primitive

Simple primitive that call an update method at a predefined frequency.

You should write your own subclass where you only defined the `update()` method.

recent_update_frequencies
Returns the 10 most recent update frequencies.

The given frequencies are computed as short-term frequencies! The 0th element of the list corresponds to the most recent frequency.

run()
Calls the `update()` method at a predefined frequency (runs until stopped).

update()
Update methods that will be called at a predefined frequency.

class pypot.primitive.primitive.**MockupRobot**(*robot*)
Bases: object

Fake `Robot` used by the `Primitive` to ensure sandboxing.

goto_position(*position_for_motors*, *duration*, *control=None*, *wait=False*)

motors
List of all attached `MockupMotor`.

power_max()

class pypot.primitive.primitive.**MockupMotor**(*motor*)
Bases: object

Fake Motor used by the primitive to ensure sandboxing:

- the read instructions are directly delegate to the real motor
- the write instructions are stored as request waiting to be combined by the primitive manager.

goto_position(*position*, *duration*, *control=None*, *wait=False*)
Automatically sets the goal position and the moving speed to reach the desired position within the duration.

goal_speed
Goal speed (in degrees per second) of the motor.

This property can be used to control your motor in speed. Setting a goal speed will automatically change the moving speed and sets the goal position as the angle limit.

Note: The motor will turn until reaching the angle limit. But this is not a wheel mode, so the motor will stop at its limits.

---

pypot.primitive.utils module

class pypot.primitive.utils.**Sinus**(*robot*, *refresh_freq*, *motor_list*, *amp=1*, *freq=0.5*, *offset=0*, *phase=0*)
Bases: pypot.primitive.primitive.LoopPrimitive

Apply a sinus on the motor specified as argument. Parameters (amp, offset and phase) should be specified in degree.

properties = ['frequency', 'amplitude', 'offset', 'phase']

update()
Compute the sin(t) where t is the elapsed time since the primitive has been started.

frequency

amplitude

offset

phase

class pypot.primitive.utils.**Cosinus**(*robot*, *refresh_freq*, *motor_list*, *amp=1*, *freq=0.5*, *offset=0*, *phase=0*)
Bases: pypot.primitive.utils.Sinus

Apply a cosinus on the motor specified as argument. Parameters (amp, offset and phase) should be specified in degree.

class pypot.primitive.utils.**Square**(*robot*, *refresh_freq*, *motor_list*, *amp=1*, *freq=1.0*, *offset=0*, *phase=0*, *duty=0.5*)
Bases: pypot.primitive.utils.Sinus

Apply a square signal. Param (amp, freq, offset, phase, duty cycle).

update()

duty

class pypot.primitive.utils.**PositionWatcher**(*robot*, *refresh_freq*, *watched_motors*)
Bases: pypot.primitive.primitive.LoopPrimitive

record_positions

setup()

update()

plot(*ax*)

class pypot.primitive.utils.**SimplePosture**(*robot*, *duration*)
Bases: pypot.primitive.primitive.Primitive

setup()

run()

teardown()

Module contents

pypot.robot package

Submodules

pypot.robot.config module The config module allows the definition of the structure of your robot.

Configuration are written as Python dictionary so you can define/modify them programmatically. You can also import them form file such as JSON formatted file. In the configuration you have to define:

- controllers: For each defined controller, you can specify the port name, the attached motors and the synchronization mode.
- motors: You specify all motors belonging to your robot. You have to define their id, type, orientation, offset and angle_limit.
- motorgroups: It allows to define alias of group of motors. They can be nested.

pypot.robot.config.**from_config**(*config*, *strict=True*, *sync=True*, *use_dummy_io=False*)
Returns a Robot instance created from a configuration dictionnary.

Parameters

- **config** (*dict*) – robot configuration dictionary
- **strict** (*bool*) – make sure that all ports, motors are availaible.
- **sync** (*bool*) – choose if automatically starts the synchronization loops

For details on how to write such a configuration dictionnary, you should refer to the section config_file.

pypot.robot.config.**motor_from_configgnode**(*config*, *motor_name*)

pypot.robot.config.**sensor_from_configgnode**(*config*, *s_name*, *robot*)

pypot.robot.config.**dxl_io_from_configgnode**(*config*, *c_params*, *ids*, *strict*)

pypot.robot.config.**check_motor_limits**(*config*, *dxl_io*, *motor_names*)

pypot.robot.config.**instatiate_motors**(*config*)

pypot.robot.config.**make_alias**(*config*, *robot*)

pypot.robot.config.**from_json**(*json_file*, *sync=True*, *strict=True*, *use_dummy_io=False*)
Returns a Robot instance created from a JSON configuration file.

For details on how to write such a configuration file, you should refer to the section config_file.

pypot.robot.config.**use_dummy_robot**(*json_file*)

pypot.robot.controller module

class pypot.robot.controller.**AbstractController**(*io*, *sync_freq*)
Bases: pypot.utils.stoppablethread.StoppableLoopThread

Abstract class for motor/sensor controller.

The controller role is to synchronize the reading/writing of a set of instances with their "hardware" equivalent through an `AbstractIO` object. It is defined as a `StoppableLoopThread` where each loop update synchronizes values from the "software" objects with their "hardware" equivalent.

To define your Controller, you need to define the `update()` method. This method will be called at the predefined frequency. An exemple of how to do it can be found in `BaseDxlController`.

Parameters

- **io** (`AbstractIO`) – IO used to communicate with the hardware motors
- **sync_freq** (*float*) – synchronization frequency

start()

close()
Cleans and closes the controller.

class pypot.robot.controller.**MotorsController**(*io*, *motors*, *sync_freq=50*)
Bases: pypot.robot.controller.AbstractController

Abstract class for motors controller.

The controller synchronizes the reading/writing of a set of motor instances with their "hardware". Each update loop synchronizes values from the "software" *DxlMotor* with their "hardware" equivalent.

Parameters

- **io** (`AbstractIO`) – IO used to communicate with the hardware motors
- **motors** (*list*) – list of motors attached to the controller
- **sync_freq** (*float*) – synchronization frequency

class pypot.robot.controller.**DummyController**(*motors*)
Bases: pypot.robot.controller.MotorsController

update()

class pypot.robot.controller.**SensorsController**(*io*, *sensors*, *sync_freq=50.0*)
Bases: pypot.robot.controller.AbstractController

Abstract class for sensors controller.

The controller frequently pulls new data from a "real" sensor and updates its corresponding software instance.

Parameters

- **io** (`AbstractIO`) – IO used to communicate with the hardware motors
- **sensors** (*list*) – list of sensors attached to the controller
- **sync_freq** (*float*) – synchronization frequency

pypot.robot.io module

class pypot.robot.io.**AbstractIO**
Bases: object

AbstractIO class which handles communication with "hardware" motors.

close()
Clean and close the IO connection.

pypot.robot.motor module

class pypot.robot.motor.**Motor**(*name*)

Bases: object

Purely abstract class representing any motor object.

registers = []

name

pypot.robot.remote module

class pypot.robot.remote.**RemoteRobotClient**(*host*, *port*)

Bases: object

Remote Access to a Robot through the REST API.

This RemoteRobot gives you access to motors and alias. For each motor you can read/write all of their registers.

You also have access to primitives. More specifically you can start/stop them.

pypot.robot.remote.**from_remote**(*host*, *port*)

Remote access to a Robot through the REST API.

pypot.robot.robot module

class pypot.robot.robot.**Robot**(*motor_controllers=[]*, *sensor_controllers=[]*, *sync=True*)

Bases: object

This class is used to regroup all motors and sensors of your robots.

Most of the time, you do not want to directly instantiate this class, but you rather want to use a factory which creates a robot instance - e.g. from a python dictionnary (see config_file).

This class encapsulates the different controllers (such as dynamixel ones) that automatically synchronize the virtual sensors/effectors instances held by the robot class with the real devices. By doing so, each sensor/effector can be synchronized at a different frequency.

This class also provides a generic motors accessor in order to (more or less) easily extends this class to other types of motor.

Parameters

- **motor_controllers** (*list*) – motors controllers to attach to the robot
- **sensor_controllers** (*list*) – sensors controllers to attach to the robot
- **sync** (*bool*) – choose if automatically starts the synchronization loops

close()

Cleans the robot by stopping synchronization and all controllers.

start_sync()

Starts all the synchonization loop (sensor/effector controllers).

stop_sync()

Stops all the synchonization loop (sensor/effector controllers).

attach_primitive(*primitive*, *name*)

motors

Returns all the motors attached to the robot.

sensors

Returns all the sensors attached to the robot.

active_primitives
Returns all the primitives currently running on the robot.

primitives
Returns all the primitives name attached to the robot.

compliant
Returns a list of all the compliant motors.

goto_position(*position_for_motors*, *duration*, *control=None*, *wait=False*)
Moves a subset of the motors to a position within a specific duration.

Parameters

- **position_for_motors** (*dict*) – which motors you want to move {motor_name: pos, motor_name: pos,...}
- **duration** (*float*) – duration of the move
- **control** (*str*) – control type ('dummy', 'minjerk')
- **wait** (*bool*) – whether or not to wait for the end of the move

power_up()
Changes all settings to guarantee the motors will be used at their maximum power.

to_config()
Generates the config for the current robot.

Note: The generated config should be used as a basis and must probably be modified.

pypot.robot.sensor module

class pypot.robot.sensor.**Sensor**(*name*)
Bases: object

Purely abstract class representing any sensor object.

registers = []

name

class pypot.robot.sensor.**ObjectTracker**(*name*)
Bases: pypot.robot.sensor.Sensor

registers = ['position', 'orientation']

position

orientation

Module contents

pypot.sensor package

Subpackages

pypot.sensor.camera package

Submodules

pypot.sensor.camera.abstractcam module

class pypot.sensor.camera.abstractcam.**AbstractCamera**(*name*, *resolution*, *fps*)

Bases: pypot.robot.sensor.Sensor

registers = ['frame', 'resolution', 'fps']

frame

post_processing(*image*)

grab()

resolution

fps

pypot.sensor.camera.opencvcam module

pypot.sensor.camera.rpicam module

Module contents

class pypot.sensor.camera.**CameraController**(*camera*)

Bases: pypot.robot.controller.SensorsController

pypot.sensor.imagefeature package

Submodules

pypot.sensor.imagefeature.blob module

pypot.sensor.imagefeature.face module

pypot.sensor.imagefeature.marker module

Module contents

pypot.sensor.kinect package

Submodules

pypot.sensor.kinect.sensor module This code has been developed by Baptiste Busch: https://github.com/buschbapti

This module allows you to retrieve Skeleton information from a Kinect device. It is only the client side of a zmq client/server application.

The server part can be found at: https://bitbucket.org/buschbapti/kinectserver/src It used the Microsoft Kinect SDK and thus only work on Windows.

Of course, the client side can be used on any platform.

class pypot.sensor.kinect.sensor.**Skeleton**
Bases: pypot.sensor.kinect.sensor.Skeleton

joints = ('hip_center', 'spine', 'shoulder_center', 'head', 'shoulder_left', 'elbow_left', 'wrist_left', 'hand_left', 'shoulde

class pypot.sensor.kinect.sensor.**Joint**(*position*, *orientation*, *pixel_coordinate*)
Bases: tuple

orientation
Alias for field number 1

pixel_coordinate
Alias for field number 2

position
Alias for field number 0

class pypot.sensor.kinect.sensor.**KinectSensor**(*addr*, *port*)
Bases: object

remove_user(*user_index*)

remove_all_users()

tracked_skeleton

get_skeleton()

run()

Module contents

Submodules

pypot.sensor.optibridge module

class pypot.sensor.optibridge.**OptiBridgeServer**(*bridge_host*, *bridge_port*, *opti_addr*, *opti_port*, *obj_name*)
Bases: threading.Thread

run()

class pypot.sensor.optibridge.**OptiTrackClient**(*bridge_host*, *bridge_port*, *obj_name*)
Bases: threading.Thread

run()

tracked_objects

recent_tracked_objects

pypot.sensor.optitrack module

class pypot.sensor.optitrack.**TrackedObject**(*position*, *quaternion*, *orientation*, *timestamp*)

Bases: tuple

orientation
Alias for field number 2

position
Alias for field number 0

quaternion
Alias for field number 1

timestamp
Alias for field number 3

pypot.sensor.optitrack.**quat2euler**(*q*)

Module contents

pypot.server package

Submodules

pypot.server.httpserver module

class pypot.server.httpserver.**MyJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

JSONEncoder which tries to call a json property before using the enconding default function.

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, long, float or None. If skipkeys is True, such items are simply skipped.

If *ensure_ascii* is true (the default), all non-ASCII characters in the output are escaped with uXXXX sequences, and the results are str instances consisting of ASCII characters only. If ensure_ascii is False, a result may be a unicode instance. This usually happens if the input contains unicode strings or the *encoding* parameter is used.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation. Since the default item separator is ‘, ‘, the output might include trailing whitespace when indent is specified. You can use separators=(‘,’, ‘: ‘) to avoid this.

If specified, separators should be a (item_separator, key_separator) tuple. The default is (', ', ': '). To get the most compact JSON representation you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

If encoding is not None, then all input strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

default(*obj*)

class pypot.server.httpserver.**EnableCors**(*origin='*'*)

Bases: object

Enable CORS (Cross-Origin Resource Sharing) headers

name = 'enable_cors'

api = 2

apply(*fn*, *context*)

class pypot.server.httpserver.**HTTPRobotServer**(*robot*, *host*, *port*, *cross_domain_origin='*'*, *quiet=True*)

Bases: *pypot.server.server.AbstractServer*

Bottle based HTTPServer used to remote access a robot.

Please refer to the REST API for an exhaustive list of the possible routes.

run(*quiet=None*, *server='tornado'*)

Start the bottle server, run forever.

pypot.server.rest module

class pypot.server.rest.**RESTRobot**(*robot*)

Bases: object

REST API for a Robot.

Through the REST API you can currently access:

- the motors list (and the aliases)
- the registers list for a specific motor
- read/write a value from/to a register of a specific motor
- the sensors list
- the registers list for a specific motor
- read/write a value from/to a register of a specific motor
- the primitives list (and the active)
- start/stop primitives

get_motors_list(*alias='motors'*)

get_motor_registers_list(*motor*)

get_registers_list(*motor*)

get_motor_register_value(*motor*, *register*)

get_register_value(*motor*, *register*)

set_motor_register_value(*motor*, *register*, *value*)

set_register_value(*motor*, *register*, *value*)

get_motors_alias()

set_goto_position_for_motor(*motor*, *position*, *duration*)

get_sensors_list()

get_sensors_registers_list(*sensor*)

get_sensor_register_value(*sensor*, *register*)

set_sensor_register_value(*sensor*, *register*, *value*)

get_primitives_list()

get_running_primitives_list()

start_primitive(*primitive*)

stop_primitive(*primitive*)

pause_primitive(*primitive*)

resume_primitive(*primitive*)

get_primitive_properties_list(*primitive*)

get_primitive_property(*primitive*, *property*)

set_primitive_property(*primitive*, *property*, *value*)

get_primitive_methods_list(*primitive*)

call_primitive_method(*primitive*, *method*, *kwargs*)

start_move_recorder(*move_name*, *motors_name=None*)

attach_move_recorder(*move_name*, *motors_name*)

get_move_recorder_motors(*move_name*)

stop_move_recorder(*move_name*)
Allow more easily than stop_primitive() to save in a filename the recorcded move

start_move_player(*move_name*, *speed=1.0*, *backwards=False*)
Move player need to have a move file <move_name.record> in the working directory to play it

get_available_record_list()
Get list of json recorded movement files

remove_move_record(*move_name*)
Remove the json recorded movement file

pypot.server.server module

class pypot.server.server.**AbstractServer**(*robot*, *host*, *port*)
Bases: object

run()

class pypot.server.server.**RemoteRobotServer**(*robot*, *host*, *port*)
Bases: *pypot.server.server.AbstractServer*

run()

pypot.server.snap module

pypot.server.snap.**get_snap_user_projects_directory**()
pypot.server.snap.**find_local_ip**()

pypot.server.snap.**set_snap_server_variables**(*host*, *port*, *snap_extension='.xml'*, *path=None*)

Allow to change dynamically port and host variable in xml Snap! project file

class pypot.server.snap.**SnapRobotServer**(*robot*, *host*, *port*, *quiet=True*)

Bases: *pypot.server.server.AbstractServer*

run()

pypot.server.zmqserver module

class pypot.server.zmqserver.**ZMQRobotServer**(*robot*, *host*, *port*)

Bases: *pypot.server.server.AbstractServer*

A ZMQServer allowing remote access of a robot instance.

The server used the REQ/REP zmq pattern. You should always first send a request and then read the answer.

run()

Run an infinite REQ/REP loop.

handle_request(*request*)

Module contents

pypot.tools package

Subpackages

pypot.tools.herborist package

Submodules

pypot.tools.herborist.herborist module

pypot.tools.herborist.herborist.**get_dxl_connection**(*port*, *baudrate*, *protocol='MX'*)
pypot.tools.herborist.herborist.**release_dxl_connection**()

class pypot.tools.herborist.herborist.**HerboristApp**(*argv*)

Bases: PyQt4.QtGui.QApplication

class UpdatePortThread

Bases: PyQt4.QtCore.QThread

port_updated

run()

HerboristApp.**update_port**(*new_ports*)

HerboristApp.**update_motor_tree**(*baud_for_ids*)

HerboristApp.**start_scanning**()

HerboristApp.**abort_scanning**()

HerboristApp.**done_scanning**()

class HerboristApp.**ScanThread**(*port*, *baudrates*, *protocol*, *id_range*, *motor_tree*, *scan_progress*)
Bases: PyQt4.QtCore.QThread

done

part_done

run()

abort()

HerboristApp.**update_motor_view**()

HerboristApp.**update_motor_position**(*pos*)

HerboristApp.**motor_position_updated**(*pos*)

HerboristApp.**switch_torque**(*torque_enable*)

HerboristApp.**enable_motor_view**(*enabled*)

HerboristApp.**update_eeprom**()

class HerboristApp.**UpdateMotorThread**(*port*, *baudrate*, *protocol*, *mid*)
Bases: PyQt4.QtCore.QThread

position_updated

stop()

run()

HerboristApp.**port**

HerboristApp.**protocol**

HerboristApp.**usb_device**

HerboristApp.**baudrate**

HerboristApp.**id**

HerboristApp.**selected_motors**

HerboristApp.**ids**

pypot.tools.herborist.herborist.**main**()

Module contents

Submodules

pypot.tools.dxl_reset module Reset a dynamixel motor to "poppy" configuration.

This utility should only be used with a single motor connected to the bus. For the moment it's only working with robotis protocol v1 (AX, RX, MX motors).

To run it: $ poppy-reset-motor 42

The motor will now have the id 42, use a 1000000 baud rates, a 0µs return delay time. The angle limit are also set (by default to (-180, 180)). Its position is also set to its base position (default: 0).

For more complex use cases, see: $ poppy-reset-motor –help

pypot.tools.dxl_reset.**leave**(*msg*)

pypot.tools.dxl_reset.**almost_equal**(*a*, *b*)

pypot.tools.dxl_reset.**main**()

Module contents

pypot.utils package

Submodules

pypot.utils.appdirs module Utilities for determining application-specific dirs.

See <http://github.com/ActiveState/appdirs> for details and usage.

pypot.utils.appdirs.**user_data_dir**(*appname=None*, *appauthor=None*, *version=None*, *roaming=False*)

Return full path to the user-specific data dir for this application.

"appname" is the name of application. If None, just the system directory is returned.

"appauthor" (only used on Windows) is the name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to appname. You may pass False to disable it.

"version" is an optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.

"roaming" (boolean, default False) can be set True to use the Windows roaming appdata directory. That means that for users on a Windows network setup for roaming profiles, this user data will be sync'd on login. See <http://technet.microsoft.com/en-us/library/cc766489(WS.10).aspx> for a discussion of issues.

Typical user data directories are: Mac OS X: ~/Library/Application Support/<AppName> Unix: ~/.local/share/<AppName> # or in $XDG_DATA_HOME, if defined Win XP (not roaming): C:Documents and Settings<username>Application Data<AppAuthor><AppName> Win XP (roaming): C:Documents and Settings<username>Local SettingsApplication Data<AppAuthor><AppName> Win 7 (not roaming): C:Users<username>AppDataLocal<AppAuthor><AppName> Win 7 (roaming): C:Users<username>AppDataRoaming<AppAuthor><AppName>

For Unix, we follow the XDG spec and support $XDG_DATA_HOME. That means, by default "~/.local/share/<AppName>".

pypot.utils.appdirs.**site_data_dir**(*appname=None*, *appauthor=None*, *version=None*, *multipath=False*)

Return full path to the user-shared data dir for this application.

"appname" is the name of application. If None, just the system directory is returned.

"appauthor" (only used on Windows) is the name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to appname. You may pass False to disable it.

"version" is an optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.

"multipath" is an optional parameter only applicable to *nix which indicates that the entire list of data dirs should be returned. By default, the first item from XDG_DATA_DIRS is returned, or '/usr/local/share/<AppName>', if XDG_DATA_DIRS is not set

Typical user data directories are: Mac OS X: /Library/Application Support/<AppName> Unix: /usr/local/share/<AppName> or /usr/share/<AppName> Win XP: C:Documents and SettingsAll UsersApplication Data<AppAuthor><AppName> Vista: (Fail! "C:ProgramData" is a hidden *system* directory on Vista.) Win 7: C:ProgramData<AppAuthor><AppName> # Hidden, but writeable on Win 7.

For Unix, this is using the $XDG_DATA_DIRS[0] default.

WARNING: Do not use this on Windows. See the Vista-Fail note above for why.

pypot.utils.appdirs.**user_config_dir**(*appname=None*, *appauthor=None*, *version=None*, *roaming=False*)

Return full path to the user-specific config dir for this application.

"appname" is the name of application. If None, just the system directory is returned.

"appauthor" (only used on Windows) is the name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to appname. You may pass False to disable it.

"version" is an optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.

"roaming" (boolean, default False) can be set True to use the Windows roaming appdata directory. That means that for users on a Windows network setup for roaming profiles, this user data will be sync'd on login. See <http://technet.microsoft.com/en-us/library/cc766489(WS.10).aspx> for a discussion of issues.

Typical user data directories are: Mac OS X: same as user_data_dir Unix: ~/.config/<AppName> # or in $XDG_CONFIG_HOME, if defined Win *: same as user_data_dir

For Unix, we follow the XDG spec and support $XDG_CONFIG_HOME. That means, by deafult "~/.config/<AppName>".

pypot.utils.appdirs.**site_config_dir**(*appname=None*, *appauthor=None*, *version=None*, *multipath=False*)

Return full path to the user-shared data dir for this application.

"appname" is the name of application. If None, just the system directory is returned.

"appauthor" (only used on Windows) is the name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to appname. You may pass False to disable it.

"version" is an optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.

"multipath" is an optional parameter only applicable to *nix which indicates that the entire list of config dirs should be returned. By default, the first item from XDG_CONFIG_DIRS is returned, or '/etc/xdg/<AppName>', if XDG_CONFIG_DIRS is not set

Typical user data directories are: Mac OS X: same as site_data_dir Unix: /etc/xdg/<AppName> or $XDG_CONFIG_DIRS[i]/<AppName> for each value in

$XDG_CONFIG_DIRS

Win *: same as site_data_dir Vista: (Fail! "C:ProgramData" is a hidden *system* directory on Vista.)

For Unix, this is using the $XDG_CONFIG_DIRS[0] default, if multipath=False

WARNING: Do not use this on Windows. See the Vista-Fail note above for why.

pypot.utils.appdirs.**user_cache_dir**(*appname=None*, *appauthor=None*, *version=None*, *opinion=True*)

Return full path to the user-specific cache dir for this application.

"appname" is the name of application. If None, just the system directory is returned.

"appauthor" (only used on Windows) is the name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to appname. You may pass False to disable it.

"version" is an optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.

"opinion" (boolean) can be False to disable the appending of "Cache" to the base app data dir for Windows. See discussion below.

Typical user cache directories are: Mac OS X: ~/Library/Caches/<AppName> Unix: ~/.cache/<AppName> (XDG default) Win XP: C:Documents and Settings<username>Local SettingsApplication Data<AppAuthor><AppName>Cache Vista: C:Users<username>AppDataLocal<AppAuthor><AppName>Cache

On Windows the only suggestion in the MSDN docs is that local settings go in the *CSIDL_LOCAL_APPDATA* directory. This is identical to the non-roaming app data dir (the default returned by *user_data_dir* above). Apps typically put cache data somewhere *under* the given dir here. Some examples:

...MozillaFirefoxProfiles<ProfileName>Cache ...AcmeSuperAppCache1.0

OPINION: This function appends "Cache" to the *CSIDL_LOCAL_APPDATA* value. This can be disabled with the *opinion=False* option.

pypot.utils.appdirs.**user_log_dir**(*appname=None*, *appauthor=None*, *version=None*, *opinion=True*)

Return full path to the user-specific log dir for this application.

"appname" is the name of application. If None, just the system directory is returned.

"appauthor" (only used on Windows) is the name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to appname. You may pass False to disable it.

"version" is an optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.

"opinion" (boolean) can be False to disable the appending of "Logs" to the base app data dir for Windows, and "log" to the base cache dir for Unix. See discussion below.

Typical user cache directories are: Mac OS X: ~/Library/Logs/<AppName> Unix: ~/.cache/<AppName>/log # or under $XDG_CACHE_HOME if defined Win XP: C:Documents and Settings<username>Local SettingsApplication Data<AppAuthor><AppName>Logs Vista: C:Users<username>AppDataLocal<AppAuthor><AppName>Logs

On Windows the only suggestion in the MSDN docs is that local settings go in the *CSIDL_LOCAL_APPDATA* directory. (Note: I'm interested in examples of what some windows apps use for a logs dir.)

OPINION: This function appends "Logs" to the *CSIDL_LOCAL_APPDATA* value for Windows and appends "log" to the user cache dir for Unix. This can be disabled with the *opinion=False* option.

class pypot.utils.appdirs.**AppDirs**(*appname*, *appauthor=None*, *version=None*, *roaming=False*, *multipath=False*)

Bases: object

Convenience wrapper for getting application dirs.

user_data_dir

site_data_dir

user_config_dir

site_config_dir

user_cache_dir

user_log_dir

pypot.utils.interpolation module

class pypot.utils.interpolation.**KDTreeDict**(*gen_tree_on_add=False*, *distance_upper_bound=0.2*, *k_neighbors=2*)

Bases: dict

update(**args*, ***kwargs*)

generate_tree()

nearest_keys(*key*)

Find the nearest_keys (l2 distance) thanks to a cKDTree query

interpolate_motor_positions(*input_key*, *nearest_keys*)

Process linear interpolation to estimate actual speed and position of motors Method specific to the :meth:~pypot.primitive.move.Move.position() structure it is a KDTreeDict[timestamp] = {dict[motor]=(position,speed)}

pypot.utils.pypot_time module

pypot.utils.pypot_time.**time**()

pypot.utils.pypot_time.**sleep**(*t*)

pypot.utils.stoppablethread module

class pypot.utils.stoppablethread.**StoppableThread**(*setup=None*, *target=None*, *teardown=None*)

Bases: object

Stoppable version of python Thread.

This class provides the following mechanism on top of "classical" python Thread:

- you can stop the thread (if you defined your run method accordingly).
- you can restart a thread (stop it and re-run it)
- you can pause/resume a thread

Warning: It is up to the subclass to correctly respond to the stop, pause/resume signals (see `run()` for details).

Parameters

- **setup** (*func*) – specific setup function to use (otherwise self.setup)
- **target** (*func*) – specific target function to use (otherwise self.run)
- **teardown** (*func*) – specific teardown function to use (otherwise self.teardown)

start()
Start the run method as a new thread.

It will first stop the thread if it is already running.

stop(*wait=True*)
Stop the thread.

More precisely, sends the stopping signal to the thread. It is then up top the run method to correctly responds.

join()
Wait for the thread termination.

running
Whether the thread is running.

started
Whether the thread has been started.

wait_to_start()
Wait for the thread to actually starts.

should_stop()
Signals if the thread should be stopped or not.

wait_to_stop()
Wait for the thread to terminate.

setup()
Setup method call just before the run.

run()
Run method of the thread.

Note: In order to be stoppable (resp. pausable), this method has to check the running property - as often as possible to improve responsivness - and terminate when `should_stop()` (resp. `should_pause()`) becomes True. For instance:

```
while self.should_stop():
    do_atom_work()
    ...
```

---

teardown()
Teardown method call just after the run.

should_pause()
Signals if the thread should be paused or not.

paused

pause()
Requests the thread to pause.

resume()
Requests the thread to resume.

wait_to_resume()
Waits until the thread is resumed.

pypot.utils.stoppablethread.**make_update_loop**(*thread*, *update_func*)
Makes a run loop which calls an update function at a predefined frequency.

class pypot.utils.stoppablethread.**StoppableLoopThread**(*frequency*, *update=None*)
Bases: pypot.utils.stoppablethread.StoppableThread

LoopThread calling an update method at a pre-defined frequency.

Note: This class does not mean to be accurate. The given frequency will be approximately followed - depending for instance on CPU load - and only reached if the update method takes less time than the chosen loop period.

Params float frequency called frequency of the `update()` method

run()
Called the update method at the pre-defined frequency.

update()
Update method called at the pre-defined frequency.

pypot.utils.trajectory module

class pypot.utils.trajectory.**MinimumJerkTrajectory**(*initial*, *final*, *duration*, *init_vel=0.0*, *init_acc=0.0*, *final_vel=0.0*, *final_acc=0.0*)
Bases: object

compute()

get_value(*t*)

domain(*x*)

test_domain(*x*)

fix_input(*x*)

get_generator()

class pypot.utils.trajectory.**GotoMinJerk**(*motor*, *position*, *duration*, *frequency=50*)
Bases: pypot.utils.stoppablethread.StoppableLoopThread

setup()

update()

elapsed_time

Module contents

class pypot.utils.**Point2D**(*x*, *y*)
Bases: tuple

x
Alias for field number 0

y
Alias for field number 1

class pypot.utils.**Point3D**(*x*, *y*, *z*)
Bases: tuple

x
Alias for field number 0

y
Alias for field number 1

z
Alias for field number 2

pypot.utils.**Point**
alias of Point3D

class pypot.utils.**Vector3D**(*x*, *y*, *z*)
Bases: tuple

x
Alias for field number 0

y
Alias for field number 1

z
Alias for field number 2

pypot.utils.**Vector**
alias of Vector3D

class pypot.utils.**Quaternion**(*x*, *y*, *z*, *w*)
Bases: tuple

w
Alias for field number 3

x
Alias for field number 0

y
Alias for field number 1

z
Alias for field number 2

pypot.utils.**attrsetter**(*item*)

class pypot.utils.**SyncEvent**(*period=0.1*)
Bases: object

request()

done()

is_recent

needed

pypot.vrep package

Subpackages

pypot.vrep.remoteApiBindings package

Submodules

pypot.vrep.remoteApiBindings.vrep module

pypot.vrep.remoteApiBindings.vrep.**tbs**(*str*)

pypot.vrep.remoteApiBindings.vrep.**py3compatible**(*f*)

pypot.vrep.remoteApiBindings.vrep.**simxGetJointPosition**(*clientID*, *jointHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetJointPosition**(*clientID*, *jointHandle*, *position*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetJointMatrix**(*clientID*, *jointHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetSphericalJointMatrix**(*clientID*, *jointHandle*, *matrix*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetJointTargetVelocity**(*clientID*, *jointHandle*, *targetVelocity*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetJointTargetPosition**(*clientID*, *jointHandle*, *targetPosition*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxJointGetForce**(*clientID*, *jointHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetJointForce**(*clientID*, *jointHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetJointForce**(*clientID*, *jointHandle*, *force*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxReadForceSensor**(*clientID*, *forceSensorHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxBreakForceSensor**(*clientID*, *forceSensorHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxReadVisionSensor**(*clientID*, *sensorHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectHandle**(*clientID*, *objectName*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetVisionSensorImage**(*clientID*, *sensorHandle*, *options*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetVisionSensorImage**(*clientID*, *sensorHandle*, *image*, *options*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetVisionSensorDepthBuffer**(*clientID*, *sensorHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectChild**(*clientID*, *parentObjectHandle*, *childIndex*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectParent**(*clientID*, *childObjectHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxReadProximitySensor**(*clientID*, *sensorHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxLoadModel**(*clientID*, *modelPathAndName*, *options*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxLoadUI**(*clientID*, *uiPathAndName*, *options*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxLoadScene**(*clientID*, *scenePathAndName*, *options*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxStartSimulation**(*clientID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxPauseSimulation**(*clientID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxStopSimulation**(*clientID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetUIHandle**(*clientID*, *uiName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetUISlider**(*clientID*, *uiHandle*, *uiButtonID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetUISlider**(*clientID*, *uiHandle*, *uiButtonID*, *position*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetUIEventButton**(*clientID*, *uiHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetUIButtonProperty**(*clientID*, *uiHandle*, *uiButtonID*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetUIButtonProperty**(*clientID*, *uiHandle*, *uiButtonID*, *prop*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxAddStatusbarMessage**(*clientID*, *message*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxAuxiliaryConsoleOpen**(*clientID*, *title*, *maxLines*, *mode*, *position*, *size*, *textColor*, *backgroundColor*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxAuxiliaryConsoleClose**(*clientID*, *consoleHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxAuxiliaryConsolePrint**(*clientID*, *consoleHandle*, *txt*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxAuxiliaryConsoleShow**(*clientID*, *consoleHandle*, *showState*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectOrientation**(*clientID*, *objectHandle*, *relativeToObjectHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectPosition**(*clientID*, *objectHandle*, *relativeToObjectHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetObjectOrientation**(*clientID*, *objectHandle*, *relativeToObjectHandle*, *eulerAngles*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetObjectPosition**(*clientID*, *objectHandle*, *relativeToObjectHandle*, *position*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetObjectParent**(*clientID*, *objectHandle*, *parentObject*, *keepInPlace*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetUIButtonLabel**(*clientID*, *uiHandle*, *uiButtonID*, *upStateLabel*, *downStateLabel*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetLastErrors**(*clientID*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetArrayParameter**(*clientID*, *paramIdentifier*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetArrayParameter**(*clientID*, *paramIdentifier*, *paramValues*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetBooleanParameter**(*clientID*, *paramIdentifier*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetBooleanParameter**(*clientID*, *paramIdentifier*, *paramValue*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetIntegerParameter**(*clientID*, *paramIdentifier*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetIntegerParameter**(*clientID*, *paramIdentifier*, *paramValue*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetFloatingParameter**(*clientID*, *paramIdentifier*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetFloatingParameter**(*clientID*, *paramIdentifier*, *paramValue*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetStringParameter**(*clientID*, *paramIdentifier*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetCollisionHandle**(*clientID*, *collisionObjectName*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetDistanceHandle**(*clientID*, *distanceObjectName*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxReadCollision**(*clientID*, *collisionObjectHandle*, *operationMode*)

Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxReadDistance**(*clientID*, *distanceObjectHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxRemoveObject**(*clientID*, *objectHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxRemoveModel**(*clientID*, *objectHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxRemoveUI**(*clientID*, *uiHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxCloseScene**(*clientID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjects**(*clientID*, *objectType*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxDisplayDialog**(*clientID*, *titleText*, *mainText*, *dialogType*, *initialText*, *titleColors*, *dialogColors*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxEndDialog**(*clientID*, *dialogHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetDialogInput**(*clientID*, *dialogHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetDialogResult**(*clientID*, *dialogHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxCopyPasteObjects**(*clientID*, *objectHandles*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectSelection**(*clientID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetObjectSelection**(*clientID*, *objectHandles*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxClearFloatSignal**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxClearIntegerSignal**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxClearStringSignal**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetFloatSignal**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetIntegerSignal**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetStringSignal**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetAndClearStringSignal**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxReadStringStream**(*clientID*, *signalName*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetFloatSignal**(*clientID*, *signalName*, *signalValue*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetIntegerSignal**(*clientID*, *signalName*, *signalValue*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetStringSignal**(*clientID*, *signalName*, *signalValue*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxAppendStringSignal**(*clientID*, *signalName*, *signalValue*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxWriteStringStream**(*clientID*, *signalName*, *signalValue*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectFloatParameter**(*clientID*, *objectHandle*, *parameterID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetObjectFloatParameter**(*clientID*, *objectHandle*, *parameterID*, *parameterValue*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectIntParameter**(*clientID*, *objectHandle*, *parameterID*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetObjectIntParameter**(*clientID*, *objectHandle*, *parameterID*, *parameterValue*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetModelProperty**(*clientID*, *objectHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSetModelProperty**(*clientID*, *objectHandle*, *prop*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxStart**(*connectionAddress*, *connectionPort*, *waitUntilConnected*, *doNotReconnectOnceDisconnected*, *timeOutInMs*, *commThreadCycleInMs*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxFinish**(*clientID*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetPingTime**(*clientID*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetLastCmdTime**(*clientID*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSynchronousTrigger**(*clientID*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxSynchronous**(*clientID*, *enable*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxPauseCommunication**(*clientID*, *enable*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetInMessageInfo**(*clientID*, *infoType*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetOutMessageInfo**(*clientID*, *infoType*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetConnectionId**(*clientID*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxCreateBuffer**(*bufferSize*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxReleaseBuffer**(*buffer*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxTransferFile**(*clientID*, *filePathAndName*, *fileName_serverSide*, *timeOut*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxEraseFile**(*clientID*, *fileName_serverSide*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxCreateDummy**(*clientID*, *size*, *color*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxQuery**(*clientID*, *signalName*, *signalValue*, *retSignalName*, *timeOutInMs*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectGroupData**(*clientID*, *objectType*, *dataType*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxGetObjectVelocity**(*clientID*, *objectHandle*, *operationMode*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxPackInts**(*intList*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxUnpackInts**(*intsPackedInString*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxPackFloats**(*floatList*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrep.**simxUnpackFloats**(*floatsPackedInString*)
Please have a look at the function description/documentation in the V-REP user manual

pypot.vrep.remoteApiBindings.vrepConst module

Module contents

Submodules

pypot.vrep.controller module

class pypot.vrep.controller.**VrepController**(*vrep_io*, *scene*, *motors*, *sync_freq=50.0*)
Bases: pypot.robot.controller.MotorsController

V-REP motors controller.

Parameters

- **vrep_io** (VrepIO) – vrep io instance
- **scene** (*str*) – path to the V-REP scene file to start
- **motors** (*list*) – list of motors attached to the controller
- **sync_freq** (*float*) – synchronization frequency

setup()
Setups the controller by reading/setting position for all motors.

update()
Synchronization update loop.

At each update all motor position are read from vrep and set to the motors. The motors target position are also send to v-rep.

class pypot.vrep.controller.**VrepObjectTracker**(*io*, *sensors*, *sync_freq=50.0*)
Bases: pypot.robot.controller.SensorsController

Tracks the 3D position and orientation of a V-REP object.

Parameters

- **io** (AbstractIO) – IO used to communicate with the hardware motors
- **sensors** (*list*) – list of sensors attached to the controller
- **sync_freq** (*float*) – synchronization frequency

setup()
Forces a first update to trigger V-REP streaming.

update()
Updates the position and orientation of the tracked objects.

class pypot.vrep.controller.**VrepCollisionDetector**(*name*)
Bases: pypot.robot.sensor.Sensor

colliding

class pypot.vrep.controller.**VrepCollisionTracker**(*io*, *sensors*, *sync_freq=50.0*)
Bases: pypot.robot.controller.SensorsController

Tracks collision state.

Parameters

- **io** (AbstractIO) – IO used to communicate with the hardware motors
- **sensors** (*list*) – list of sensors attached to the controller
- **sync_freq** (*float*) – synchronization frequency

setup()
Forces a first update to trigger V-REP streaming.

update()
Update the state of the collision detectors.

pypot.vrep.io module

class pypot.vrep.io.**VrepIO**(*vrep_host='127.0.0.1'*, *vrep_port=19997*, *scene=None*, *start=False*)
Bases: pypot.robot.io.AbstractIO

This class is used to get/set values from/to a V-REP scene.

It is based on V-REP remote API (http://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm).

Starts the connection with the V-REP remote API server.

Parameters

- **vrep_host** (*str*) – V-REP remote API server host
- **vrep_port** (*int*) – V-REP remote API server port
- **scene** (*str*) – path to a V-REP scene file
- **start** (*bool*) – whether to start the scene after loading it

Warning: Only one connection can be established with the V-REP remote server API. So before trying to connect make sure that all previously started connections have been closed (see `close_all_connections()`)

MAX_ITER = 5

TIMEOUT = 0.4

open_io()

close()
Closes the current connection.

load_scene(*scene_path*, *start=False*)
Loads a scene on the V-REP server.

Parameters

- **scene_path** (*str*) – path to a V-REP scene file
- **start** (*bool*) – whether to directly start the simulation after loading the scene

Note: It is assumed that the scene file is always available on the server side.

start_simulation()
Starts the simulation.

Note: Do nothing if the simulation is already started.

Warning: if you start the simulation just after stopping it, the simulation will likely not be started. Use `restart_simulation()` instead.

restart_simulation()
Re-starts the simulation.

stop_simulation()
Stops the simulation.

pause_simulation()
Pauses the simulation.

resume_simulation()
Resumes the simulation.

get_motor_position(*motor_name*)
Gets the motor current position.

set_motor_position(*motor_name*, *position*)
Sets the motor target position.

get_motor_force(*motor_name*)

set_motor_force(*motor_name*, *force*)

get_object_position(*object_name*, *relative_to_object=None*)
Gets the object position.

set_object_position(*object_name, position=[0, 0, 0]*)
Sets the object position.

get_object_orientation(*object_name*, *relative_to_object=None*)
Gets the object orientation.

get_object_handle(*obj*)
Gets the vrep object handle.

get_collision_state(*collision_name*)
Gets the collision state.

get_collision_handle(*collision*)
Gets a vrep collisions handle.

get_simulation_current_time(*timer='CurrentTime'*)
Gets the simulation current time.

add_cube(*name*, *position*, *sizes*, *mass*)
Add Cube

add_sphere(*name, position, sizes, mass, precision=[10, 10]*)
Add Sphere

add_cylinder(*name, position, sizes, mass, precision=[10, 10]*)
Add Cylinder

add_cone(*name, position, sizes, mass, precision=[10, 10]*)
Add Cone

change_object_name(*old_name*, *new_name*)
Change object name

call_remote_api(*func_name*, **args*, ***kwargs*)
Calls any remote API func in a thread_safe way.

Parameters

- **func_name** (*str*) – name of the remote API func to call
- **args** – args to pass to the remote API call
- **kwargs** – args to pass to the remote API call

Note: You can add an extra keyword to specify if you want to use the streaming or sending mode. The oneshot_wait mode is used by default (see here for details about possible modes).

Warning: You should not pass the clientId and the operationMode as arguments. They will be automatically added.

As an example you can retrieve all joints name using the following call:

```
vrep_io.remote_api_call('simxGetObjectGroupData',
                        vrep_io.remote_api.sim_object_joint_type,
                        0,
                        streaming=True)
```

pypot.vrep.io.**close_all_connections**()
Closes all opened connection to V-REP remote API server.

exception pypot.vrep.io.**VrepIOError**(*error_code*, *message*)
Bases: exceptions.Exception

Base class for V-REP IO Errors.

exception pypot.vrep.io.**VrepIOErrors**
Bases: exceptions.Exception

exception pypot.vrep.io.**VrepConnectionError**
Bases: exceptions.Exception

Base class for V-REP connection Errors.

Module contents
class pypot.vrep.**vrep_time**(*vrep_io*)

get_time()

sleep(*t*)
pypot.vrep.**from_vrep**(*config*, *vrep_host='127.0.0.1'*, *vrep_port=19997*, *scene=None*, *tracked_objects=[]*, *tracked_collisions=[]*)
Create a robot from a V-REP instance.

Parameters

- **config** (*str or dict*) – robot configuration (either the path to the json or directly the dictionary)
- **vrep_host** (*str*) – host of the V-REP server
- **vrep_port** (*int*) – port of the V-REP server
- **scene** (*str*) – path to the V-REP scene to load and start
- **tracked_objects** (*list*) – list of V-REP dummy object to track
- **tracked_collisions** (*list*) – list of V-REP collision to track

This function tries to connect to a V-REP instance and expects to find motors with names corresponding as the ones found in the config.

Note: The Robot returned will also provide a convenience reset_simulation method which resets the simulation and the robot position to its intial stance.

Note: Using the same configuration, you should be able to switch from a real to a simulated robot just by switching from from_config() to from_vrep(). For instance:

```
import json

with open('my_config.json') as f:
    config = json.load(f)

from pypot.robot import from_config
from pypot.vrep import from_vrep

real_robot = from_config(config)
simulated_robot = from_vrep(config, '127.0.0.1', 19997, 'poppy.ttt')
```

Submodules

pypot.kinematics module

class pypot.kinematics.**Link**
Bases: *pypot.kinematics.Link*

Link object as defined by the standard DH representation.

This representation is based on the following information: :param float theta: angle about previous z from old x to new x :param float d: offset along previous z to the common normal :param float a: offset along previous to the common normal :param float alpha: angle about common normal, from old z axis to new z axis

---

Note: We are only considering revolute joint.

Please refer to http://en.wikipedia.org/wiki/Denavit-Hartenberg_parameters for more details.

get_transformation_matrix(*theta*)
Computes the homogeneous transformation matrix for this link.

class pypot.kinematics.**Chain**
Bases: *pypot.kinematics.Chain*

Chain of Link that can be used to perform forward and inverse kinematics.

Parameters

- **links** (*list*) – list of Link that compose the chain
- **base** – the base homogeneous transformation matrix
- **tool** – the end tool homogeneous transformation matrix

forward_kinematics(*q*)
Computes the homogeneous transformation matrix of the end effector of the chain.

Parameters **q** (*vector*) – vector of the joint angles (theta 1, theta 2, ..., theta n)

inverse_kinematics(*end_effector_transformation*, *q=None*, *max_iter=1000*, *tolerance=0.05*, *mask=array([1., 1., 1., 1., 1., 1.])*, *use_pinv=False*)
Computes the joint angles corresponding to the end effector transformation.

Parameters

- **end_effector_transformation** – the end effector homogeneous transformation matrix
- **q** (*vector*) – initial estimate of the joint angles
- **max_iter** (*int*) – maximum number of iteration
- **tolerance** (*float*) – tolerance before convergence
- **mask** – specify the cartesian DOF that will be ignore (in the case of a chain with less than 6 joints).

Return type vector of the joint angles (theta 1, theta 2, ..., theta n)

pypot.kinematics.**transform_difference**(*t1*, *t2*)

pypot.kinematics.**rotation_from_transf**(*tm*)

pypot.kinematics.**translation_from_transf**(*tm*)

pypot.kinematics.**components_from_transf**(*tm*)

pypot.kinematics.**transf_from_components**(*R*, *T*)

pypot.kinematics.**transl**(*x*, *y*, *z*)

pypot.kinematics.**trotx**(*theta*)

pypot.kinematics.**troty**(*theta*)

pypot.kinematics.**trotz**(*theta*)

Module contents

p

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z