Poppy project Documentation

Release 1.1.0

INRIA

October 01, 2015

CHAPTER

ONE

INTRODUCING THE POPPY PROJECT AND THE POPPY CREATURES

1.1 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.

It consist of several parts:

- Pypot, a Python library allowing to control Dynamixel servomotors in an easy way
- Poppy Humanoid, a 25-degree of freedom humanoid robot with a fully actuated torso. Used for education (high school, university), research (walk, human-robot interactions) or art (dance, performances).
- Poppy Torso, upper part of Poppy Humanoid (13 degrees of freedom). Mainly designed for school usage.
- Poppy Ergo Jr, a cheaper 6-degree-of-freedom arm for art and robot discovery.

All these parts are open-source and available to download and modify.

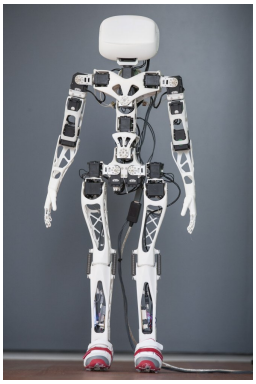
The Poppy project also consist in a very active and diverse *community*. People of different horizons collaborate to improve the project, but adding features to the core libraries, releasing advanced behaviors, pedagogical content or even new robots.

1.1.1 Poppy Humanoid

Poppy Humanoid is an open-source humanoid platform based on 3D printing.

It consists of open-source hardware models (CC-BY-SA), an open-source software library named *Pypot* (in Python, with a REST API). A simulator is also available (based on *vrep*), as well as a visual programming language (*Snap!*, a variation of Scratch). It can be used as it is, or hacked to explore the shape of novel legs, arms or hands.

From a single arm to the complete humanoid, this platform is actively used in labs, engineering schools, FabLabs, and artistic projects.



Get a Poppy Humanoid robot

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

- Génération Robots

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

Get started with Poppy Humanoid

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

1.1.2 Poppy Torso

Poppy Torso is a variation of the Poppy Humanoid robot creature: it is an open-source humanoid robot torso which can be installed easily on tabletops. 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).

Like Poppy Humanoid, Poppy Torso is an open-source robot (both hardware and software), using 3D printed parts and Dynamixel servomotors, known for their reliability.

Get a Poppy Torso

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

- Génération Robots

Or 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).

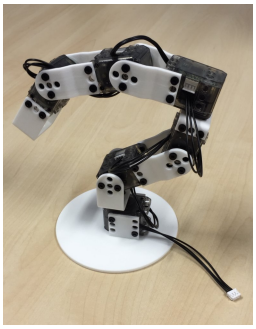
Get started with Poppy Torso

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

1.1.3 Poppy Ergo Jr

The Poppy Ergo Jr robot is a small robot arm made of 6 cheap XL-320 Dynamixel servos, 3D-printed parts based on OpenScad and assembled with OLLO rivets. At the end of the arm, you can choose among several ends: a lamp, a gripper hand,...

This robot is the 'little brother' of the ergo robots used in ' the Ergo-robot experiment <https://www.poppy-project.org/project/mathematics-a-beautiful-elsewhere/>_.



Get a Poppy Ergo Jr

Poppy Ergo Jr is still in development. However, you can already get the parts and motors, as the next development phases will probably add new pieces and not modify the existing ones.

The list of parts to print is here and the 3D files there

You then need 6 Dynamixel XL-320 (for example from here), a USB2AX to connect them to a computer and a small adaptation board.

Get started with Poppy Ergo Jr

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

1.2 Version and changelogs

1.2.1 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

1.2.2 Changelog

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

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

1.3.1 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!

1.3.2 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

1.3.3 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 witha 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.

CHAPTER

TWO

ASSEMBLY GUIDES

2.1 Dynamixel hardware

2.1.1 Introduction

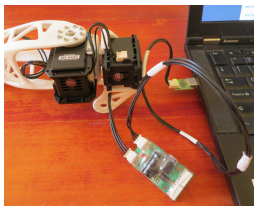
Dynamixel is a brand of 'smart' servomotors for modelism and robots. It has many different servos of different sizes and powers.

Poppy robots use only TTL (3 pin) servomotors, but there are equivalents models using the RS-485 (4 pin) communication protocol.

The Poppy Humanoid robot is mainly built with MX-28AT Dynamixel (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).

Poppy ergo jr is done from smaller XL-320. XL-320 are the latest Dynamixel servos, they use a lower voltage and a different protocol.

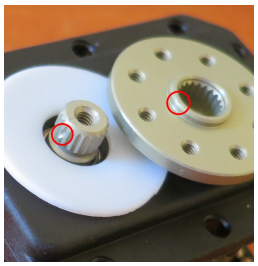
Each Dynamixel servomotor embeds an electronic board allowing it to receive different kind of orders (about goal, torque...) and communicate with other Dynamixel servos. Therefore, you can chain up several Dynamixel servomotors (each with a different ID) and command them all from one end of the chain: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



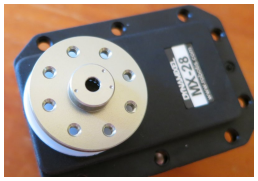
2.1.2 Putting the Dynamixel horns to zero

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.



2.1.3 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:

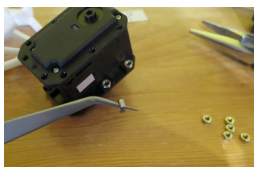
Servomotor	main horn	free horn	big horn screw	horn screws	case screws
MX64	HN05-N102	HN05-I101	M3x8mm	M2.5x4mm	M2.5x6mm
MX28	HN07-N101	HN07-I101	M2.5x8mm	M2x3mm	M2.5x6mm
AX12-A	none	none	M3x10mm	M2	M2
XL-320	none	none	none	none	none

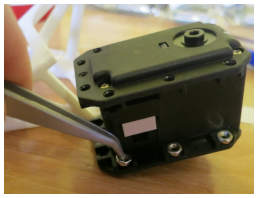
You need an allen wrench of size 1.5mm for M2 screws, 2mm for M2.5 screws and 2.5mm for M3 screws. The longer M2 screws need a Phillips screwdriver.

2.1.4 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!

2.1.5 Connection Dynamixel - computer

There are two devices allowing you to connect you 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.

USB2AX is a miniatirized version of the USB2Dynamixel able to control only TTL busses.

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

2.1.6 Powering Dynamixel servos

The USB port of your computer can't deliver enough power (well, enough current) to make your servos move.

You have to provide power (12V for MX-28, MX-64 and AX-12A, 7.4V for XL-320) through batteries or a SMPS2Dynamixel. The SMPS2Dynamixel can be used with 3-pin or 4pin motors and transmits the data from the bus.

In Poppy Humanoid and Poppy Torso, the 4-pin part of the SMPS2Dynamixel is used to bring power in the head, to the main board, audio ampli,...

2.2 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.

2.2.1 Installing the driver for USB2AX

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 !

2.2.2 Installing the scanning software

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

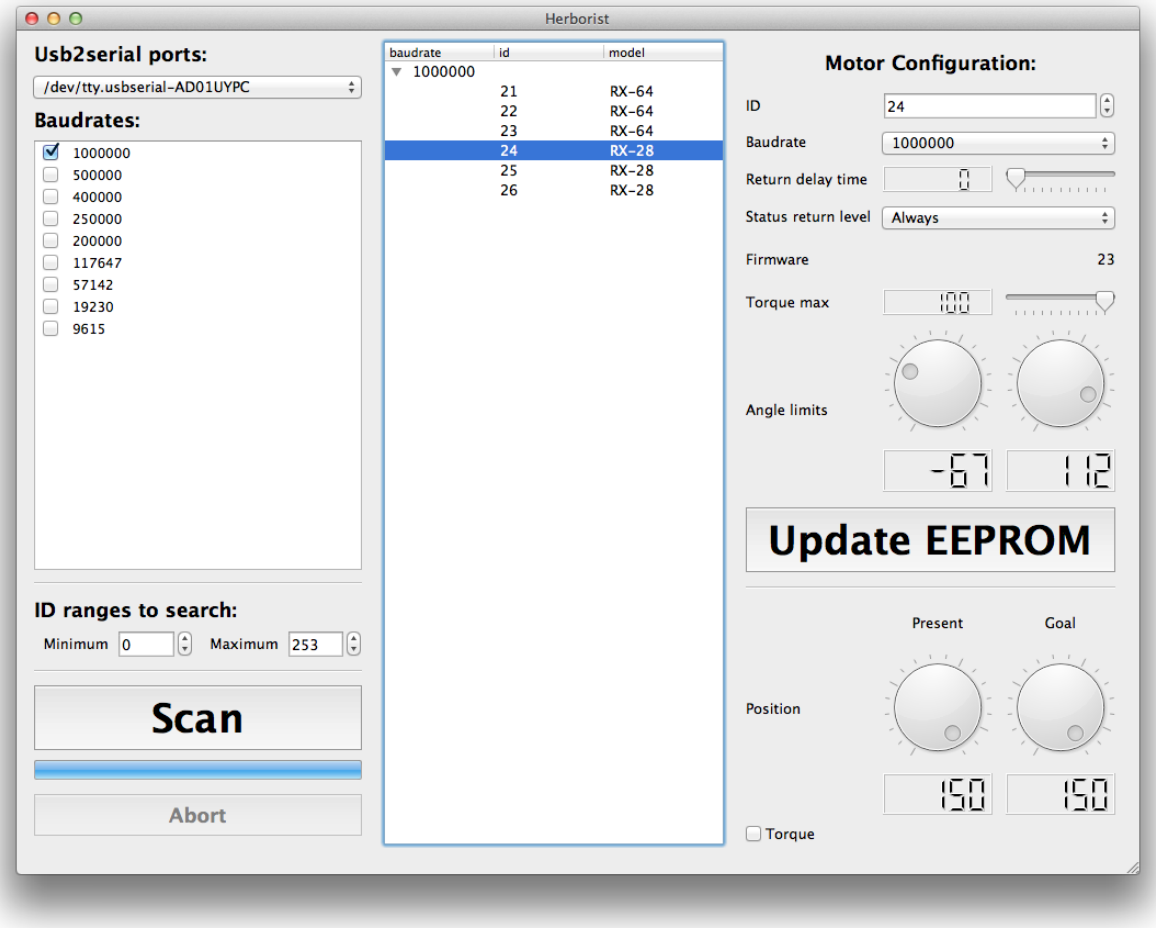
- Herborist: tool created by the Poppy Project team.
- Dynamixel Wizard: windows-only tool provided by Robotis.

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.

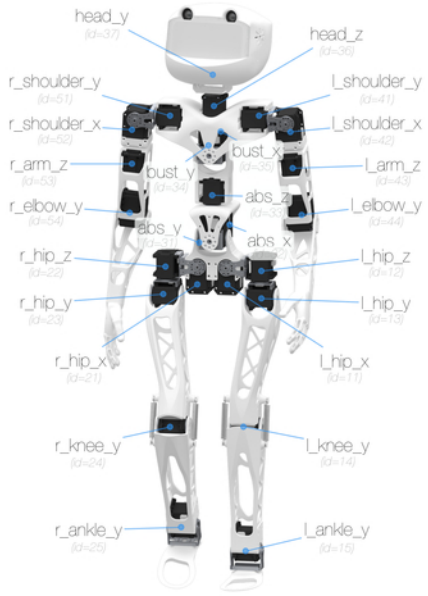
You have to set:

- 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.

2.2.3 Naming conventions

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



<< Back to menu << Dynamixel hardware

2.3 Assembly guide for a Poppy Humanoid or Poppy Torso robot

2.3.1 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

2.3.2 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

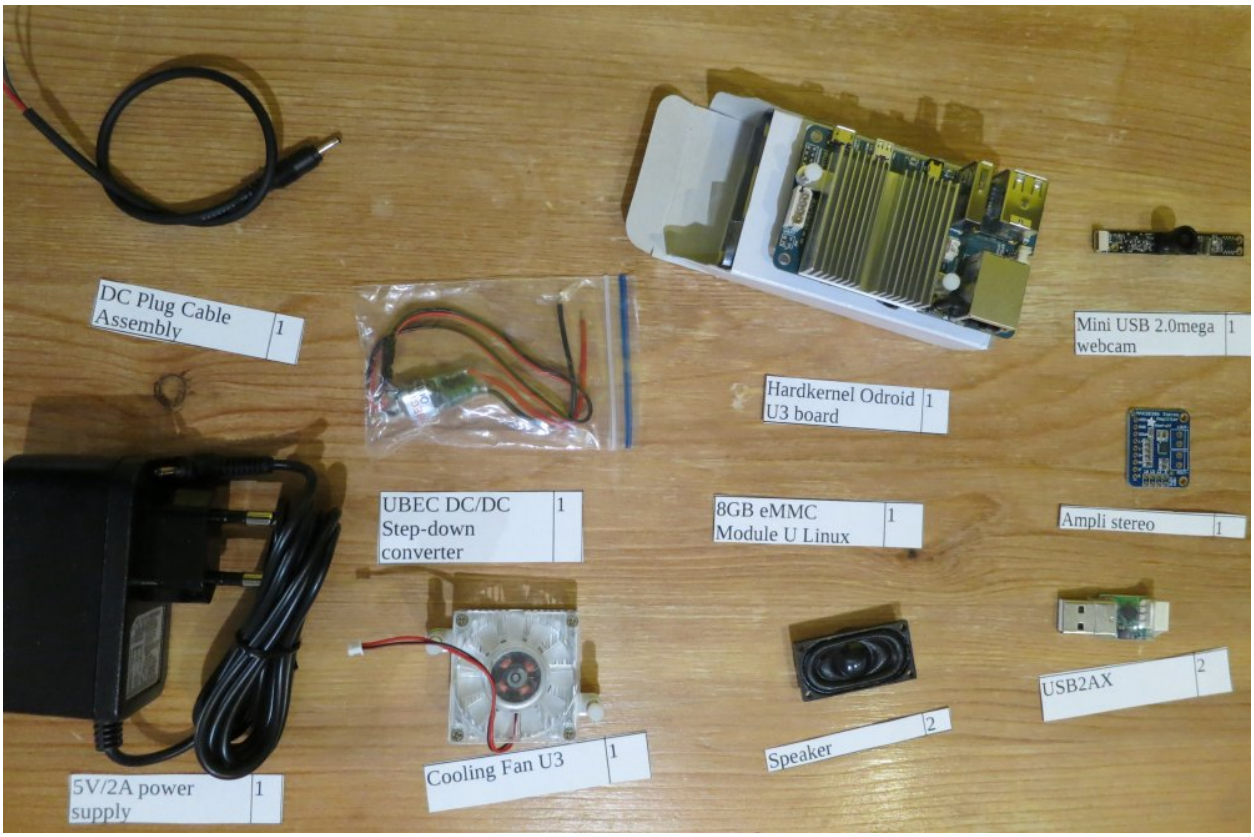
Tools

- 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

2.3.3 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:

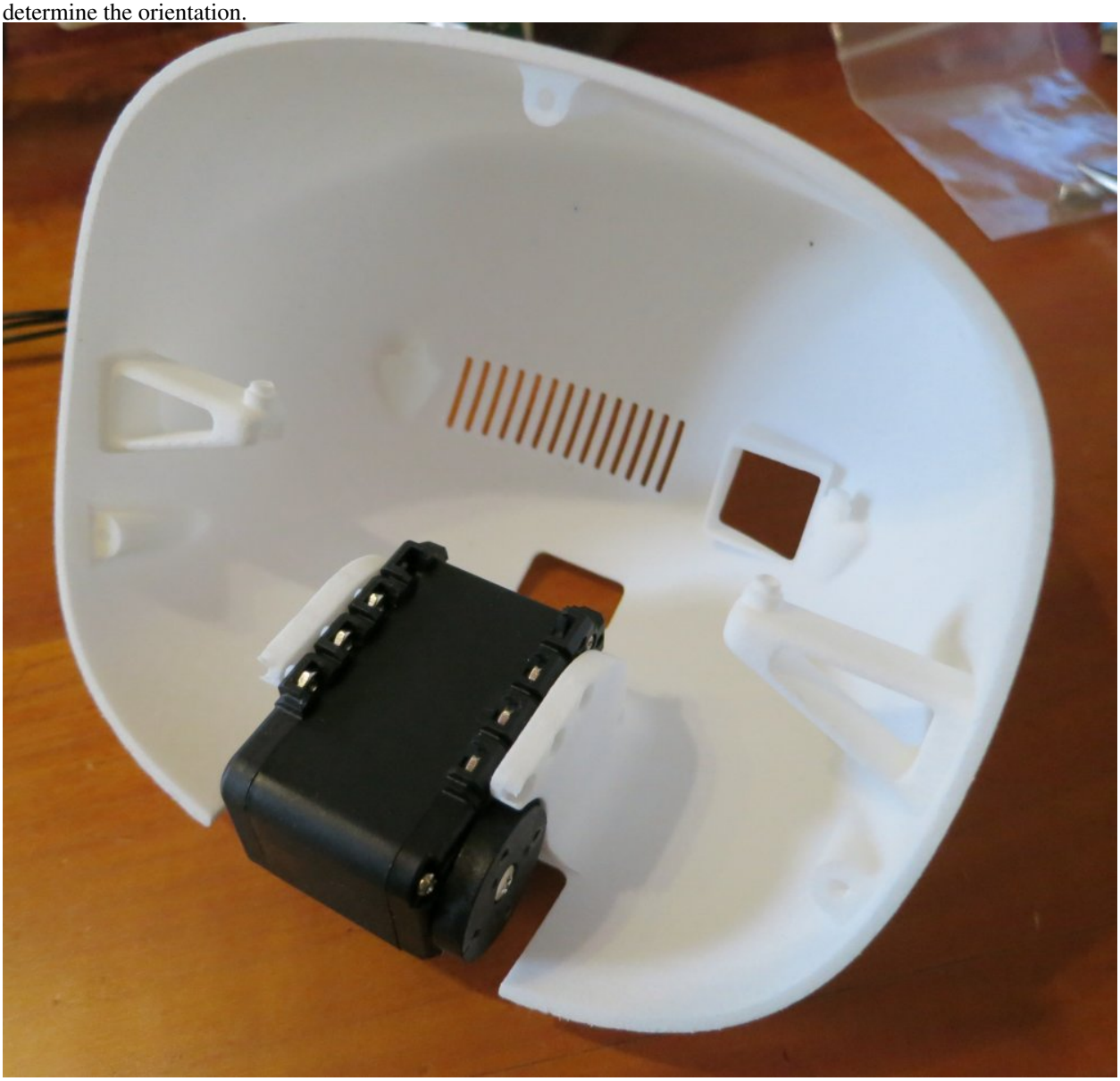
```
ssh poppy@poppy.local
```

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.

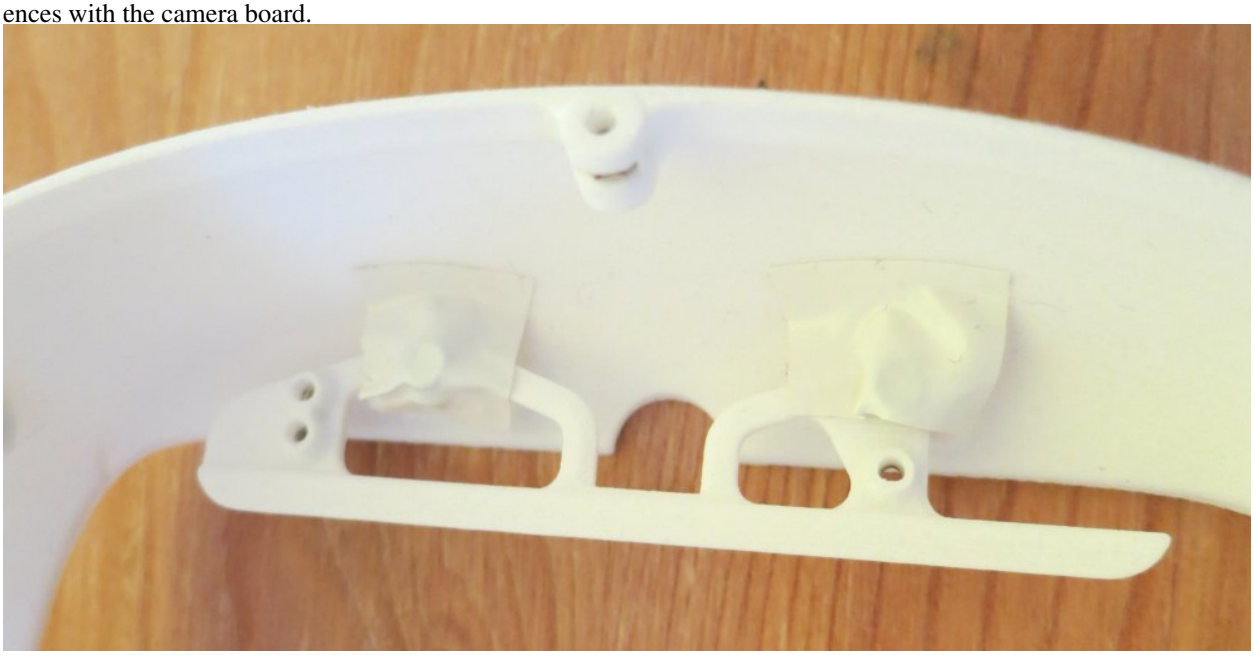
Connect head_y to the dispatcher by passing the cable through the hole in the head.

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.

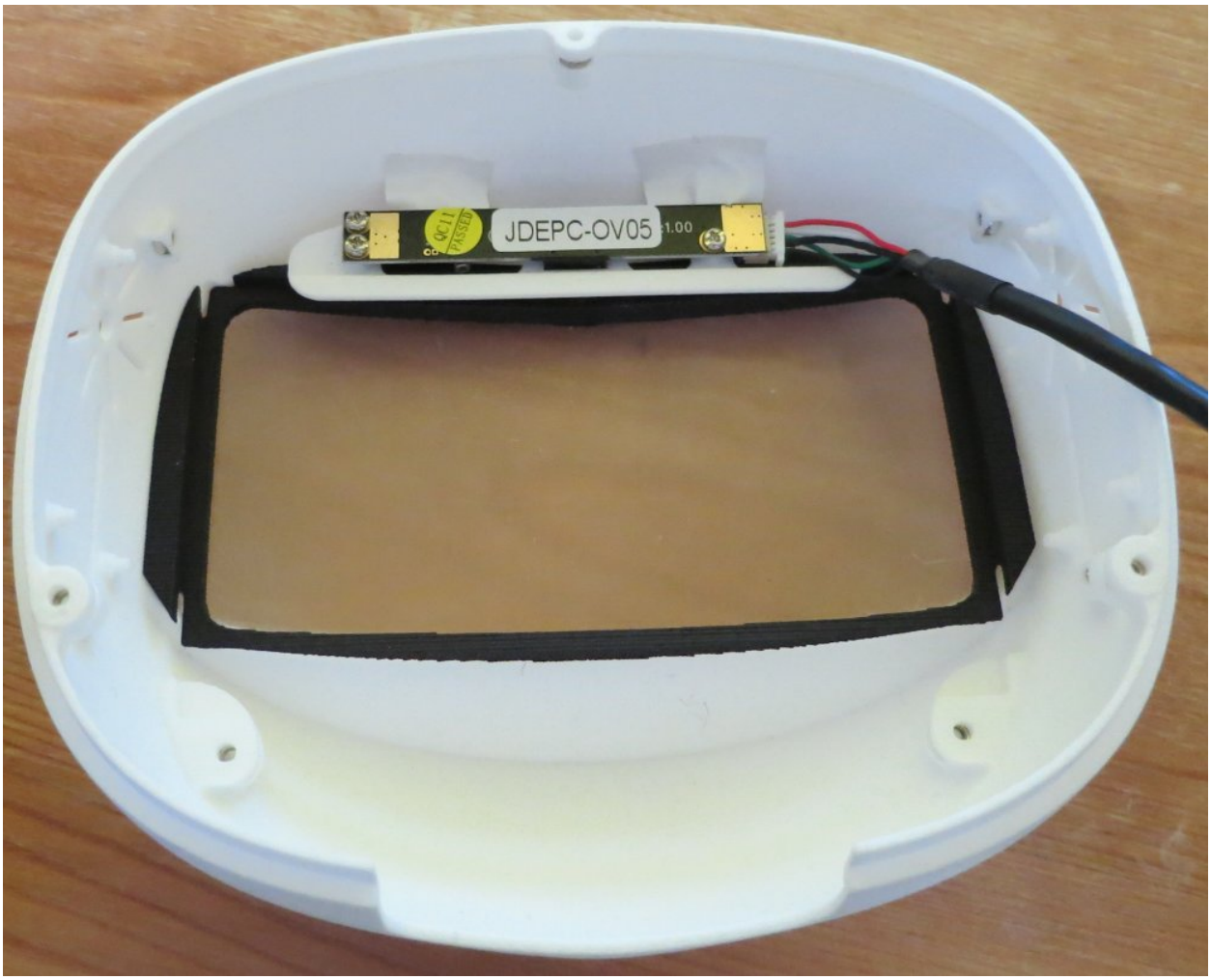
Camera and screen

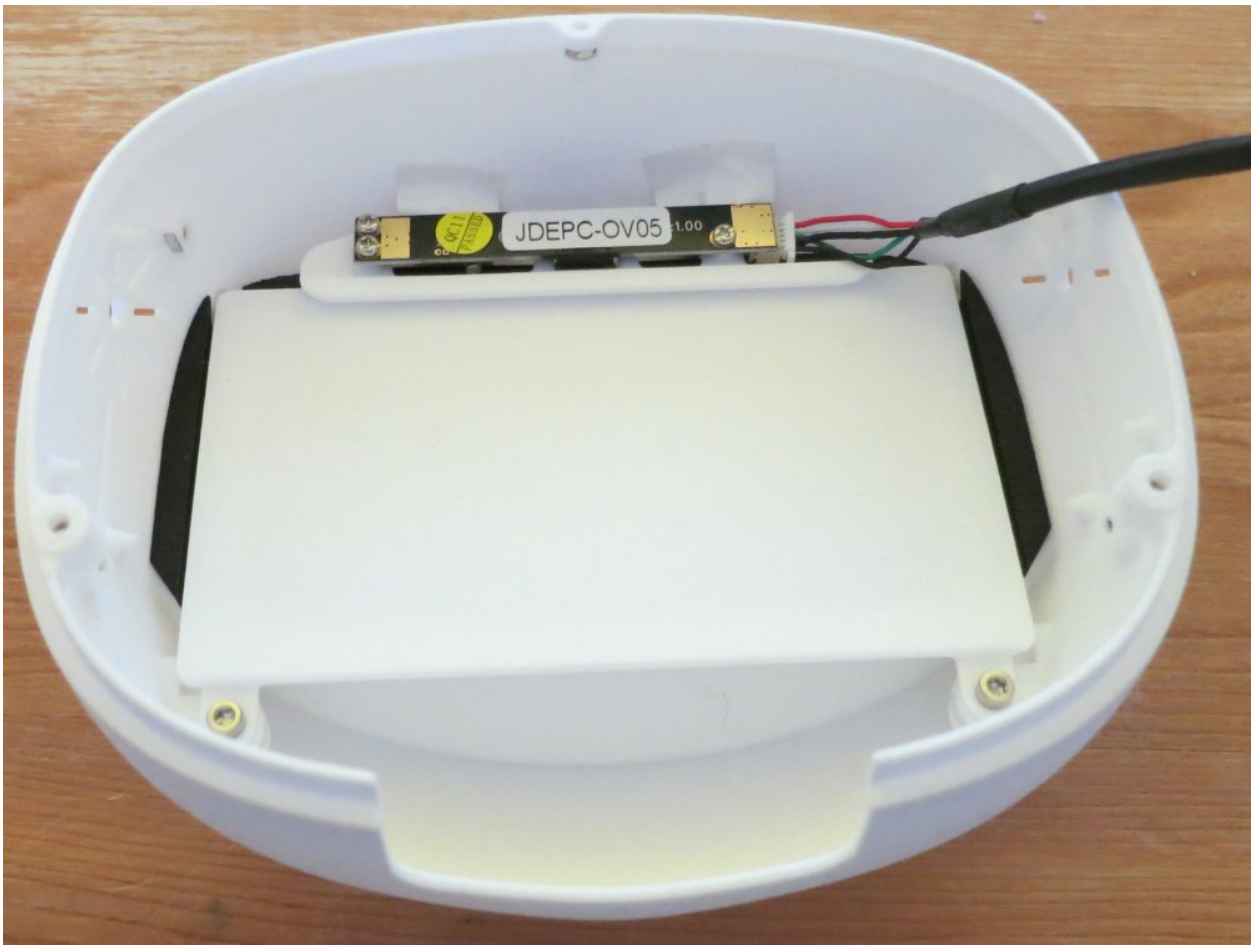
Attach the camera support to head_front using M2.5x4mm screws. Put tape on the screws to avoid electrical interferences with the camera board.



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.

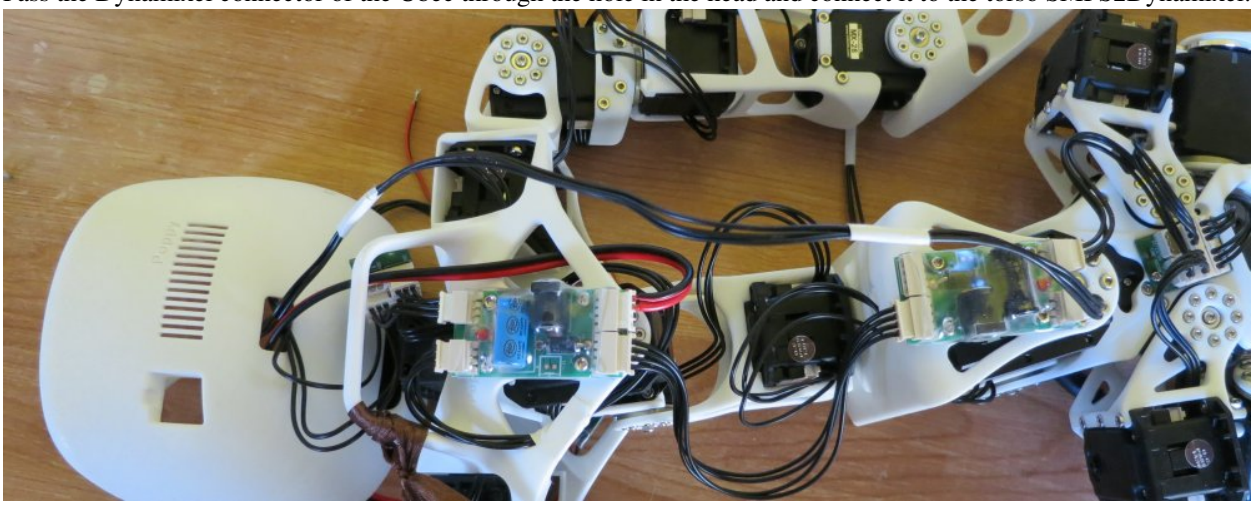

JDEPC-OV05



Electronics

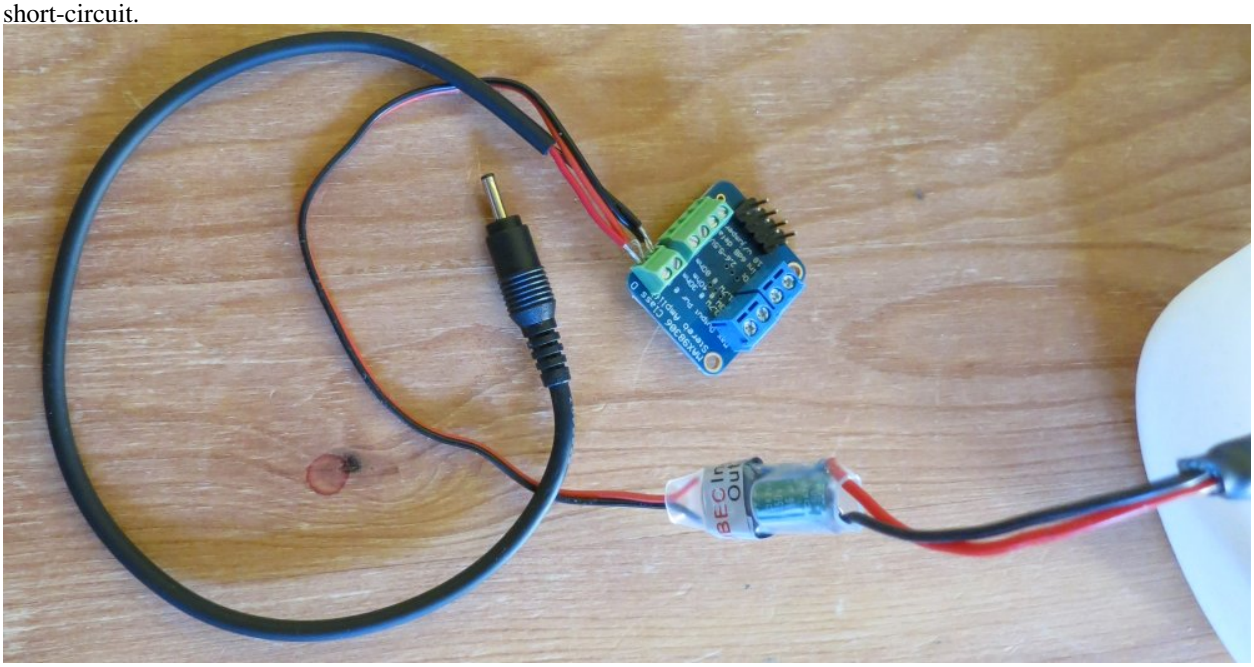
If you don't have pre-soldered components, see: https://github.com/poppy-project/Poppy-minimal-head-design/blob/master/doc/poppy_soldering.md

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

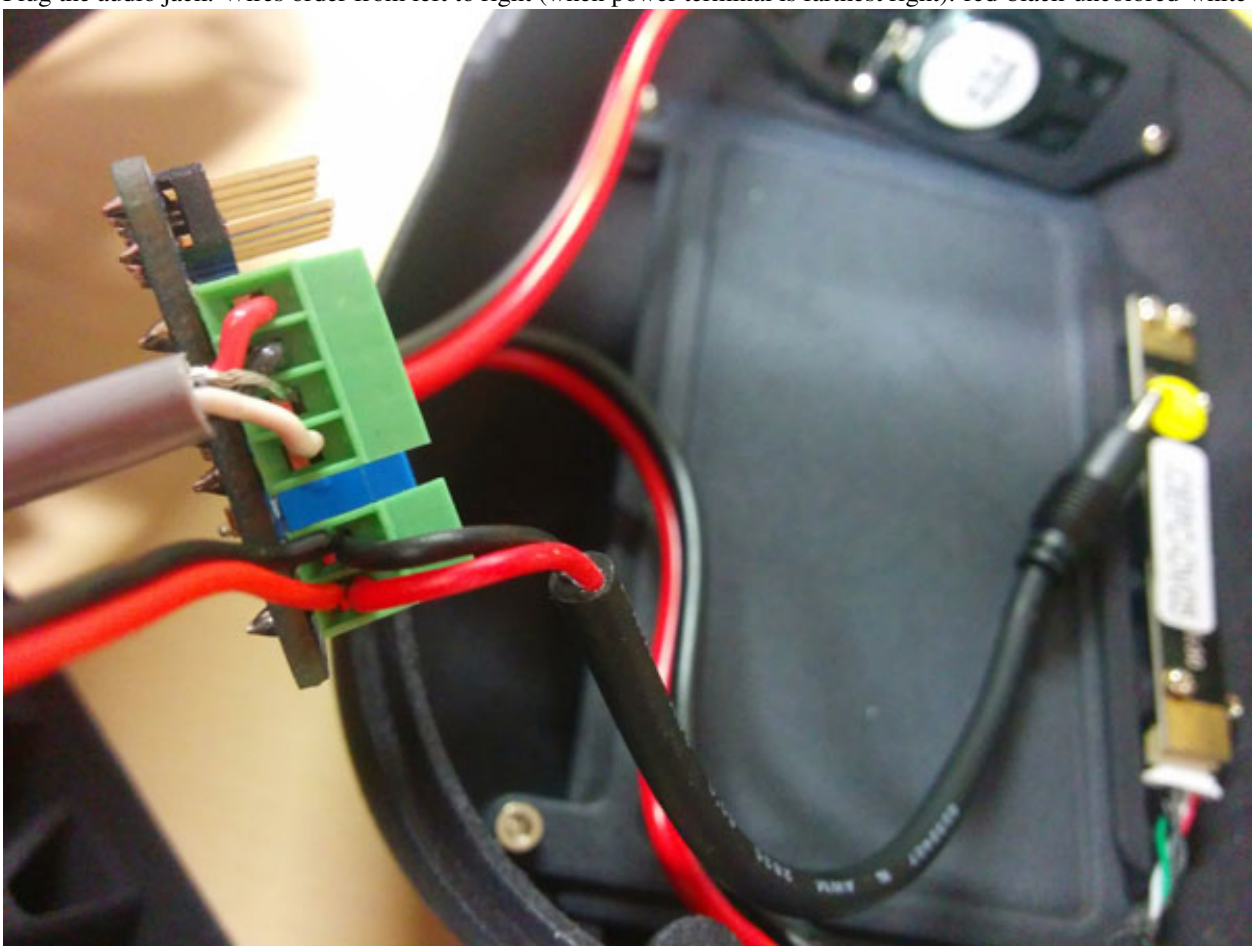


Attach both the other side of Ubec and the Odroid power cable to the audio amplifier. Be sure no to allow any

short-circuit.

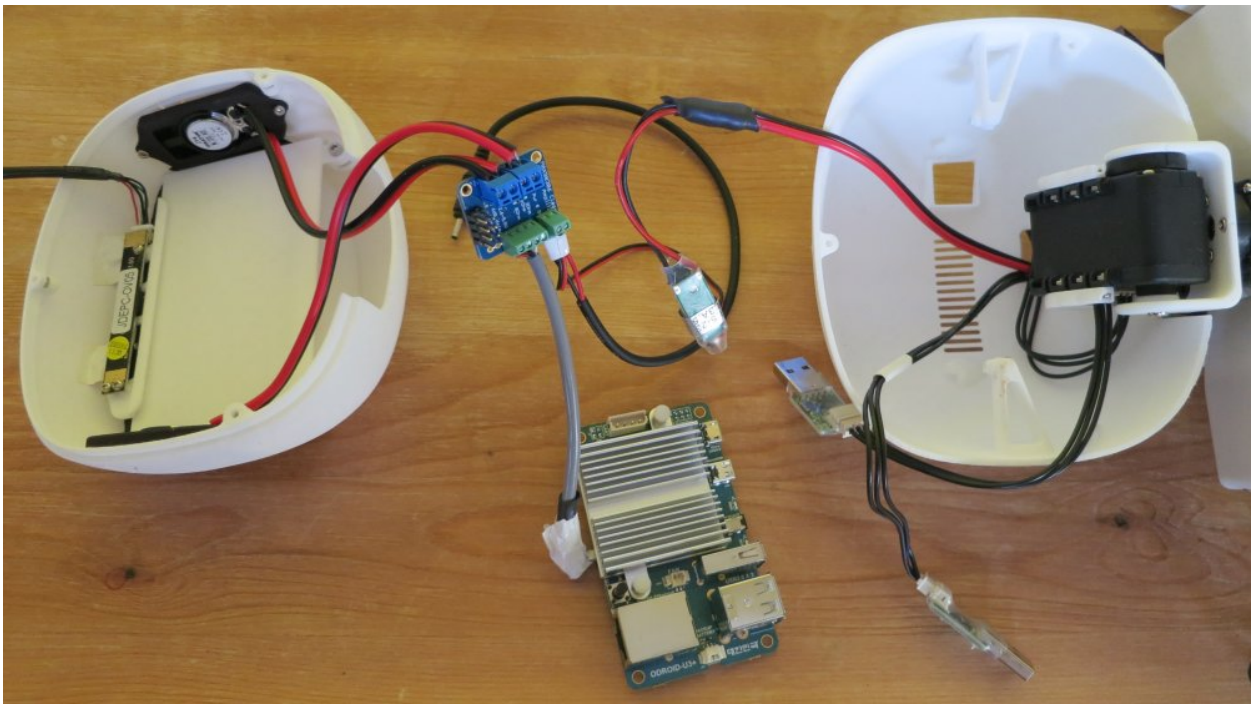


Plug the audio jack. Wires order from left to right (when power terminal is farthest right): red-black-uncolored-white

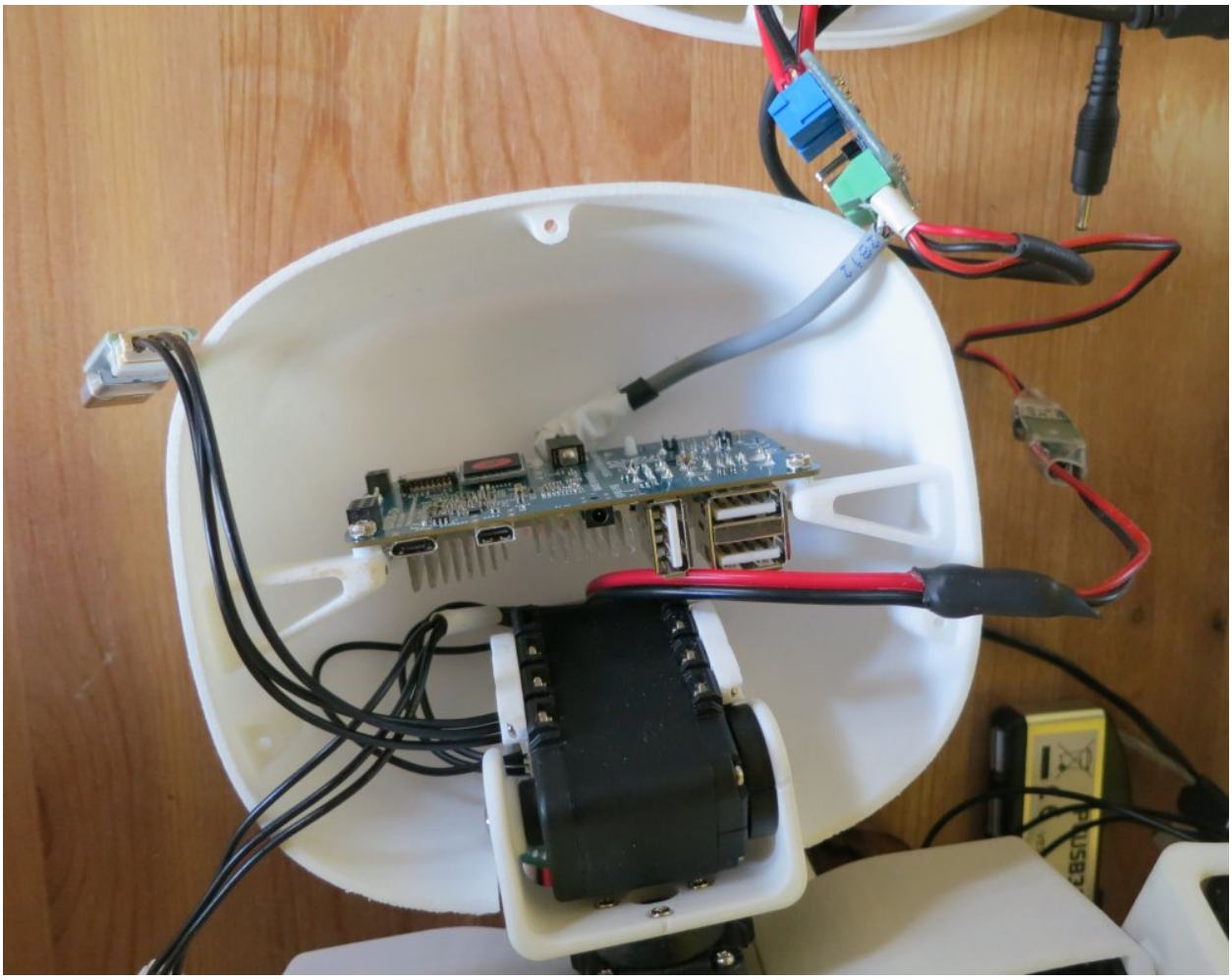


Put 2 nuts around the flowers openings then attach the speakers using M2 x3mm screws.

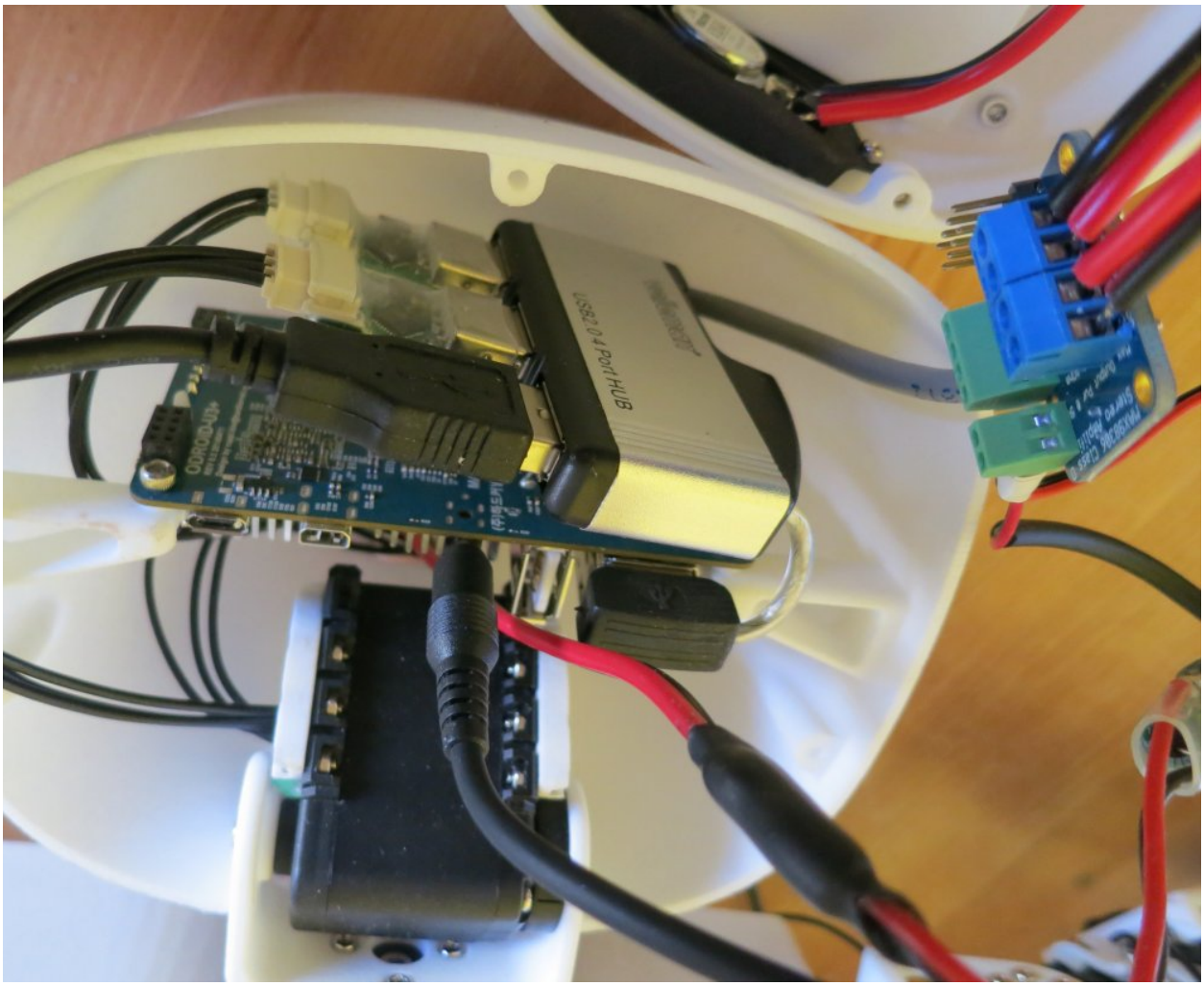
Connect the speakers to audio ampli, left speaker black wire on Lout, Right ampli black wire on Rout.



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.

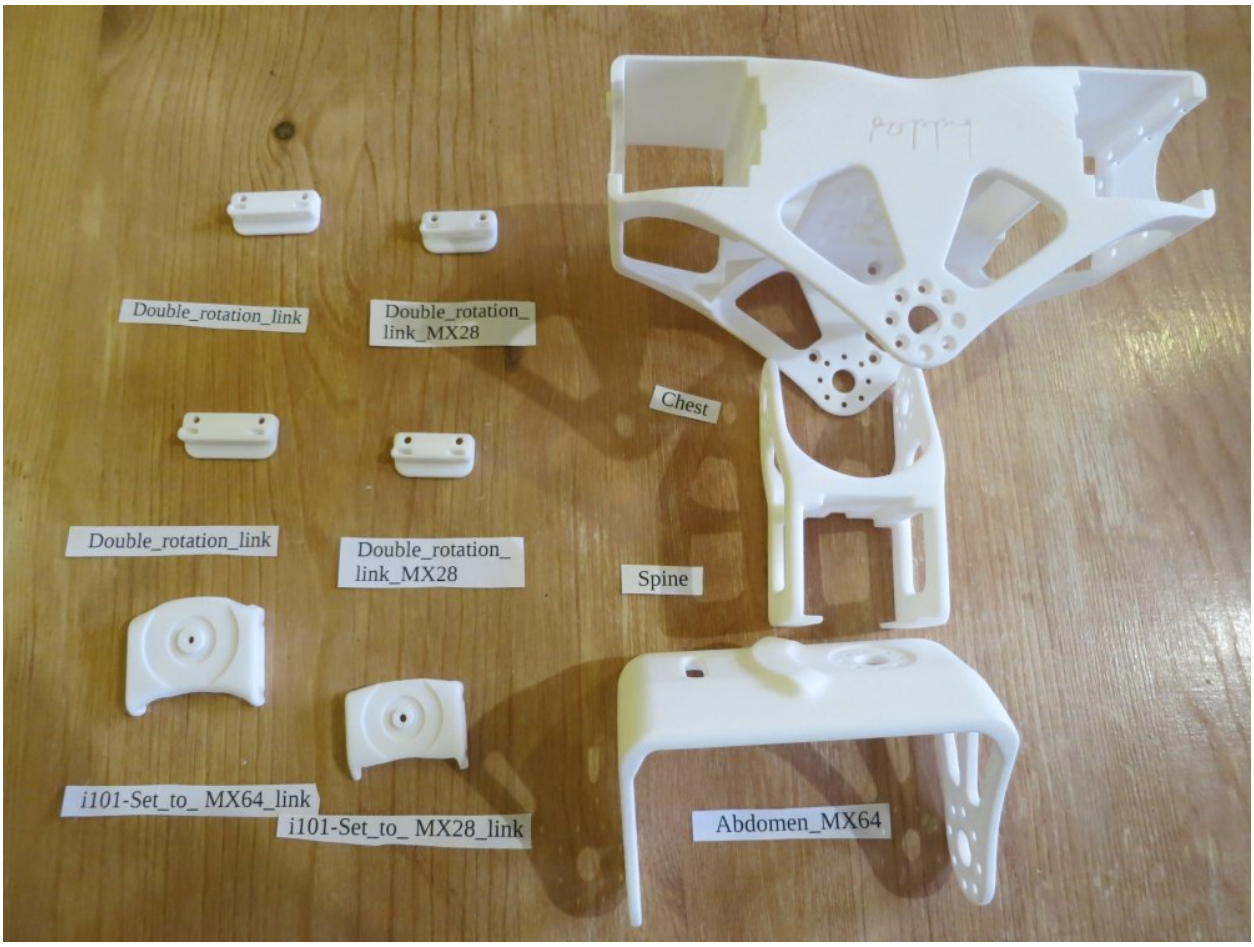


Then close the head using 3 M2x8mm screws.

<< Back to menu

Trunk assembly >>

2.3.4 Trunk



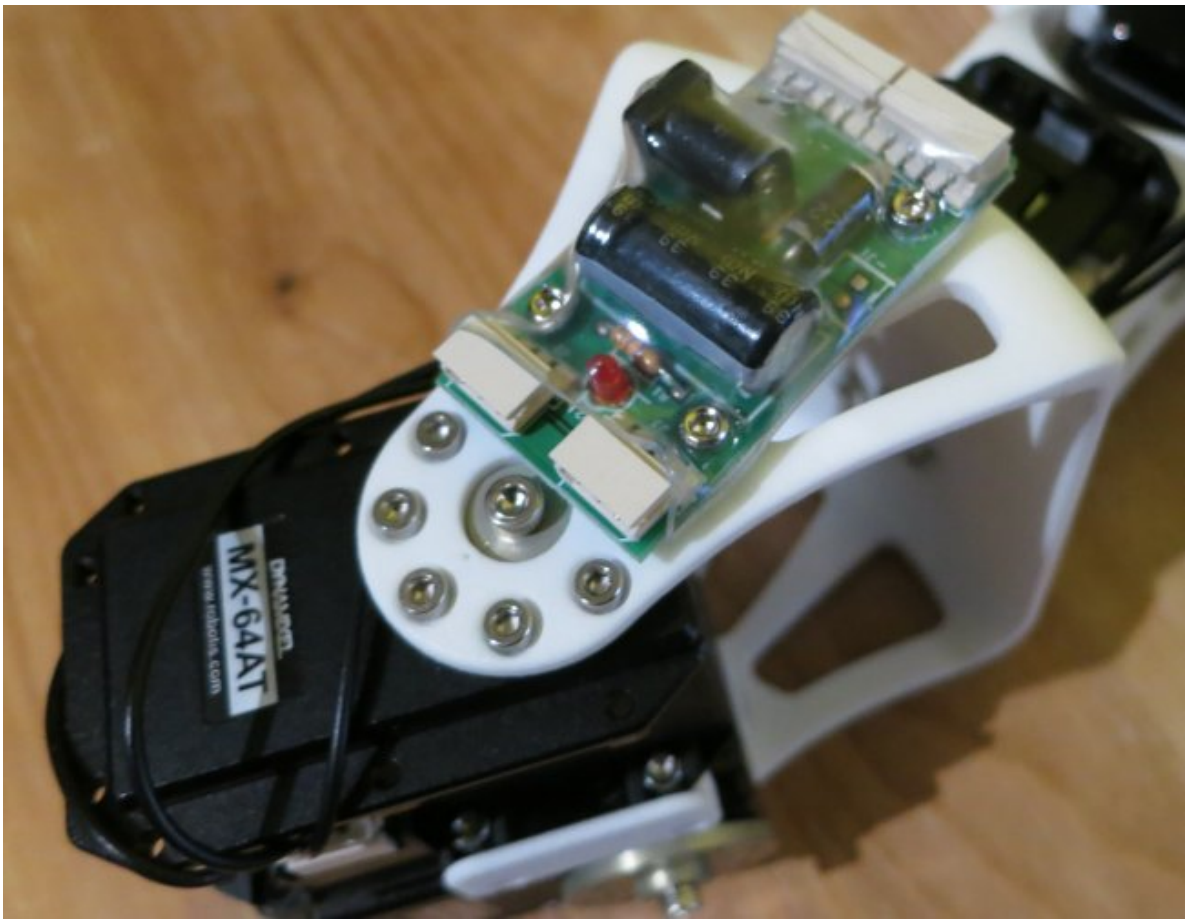
Motors list:

| 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.

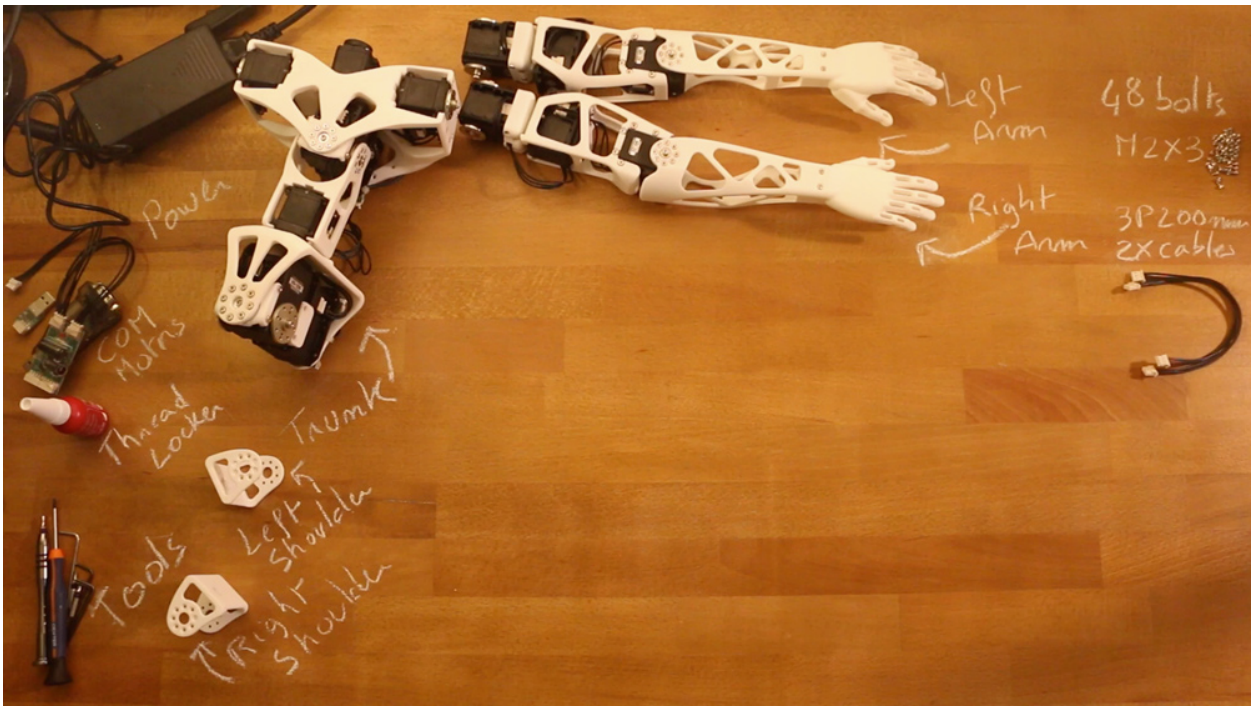
  The abdomen part you have has a "Poppy" mark on the back, while the one on the video don't. You also have holes to screw the SMPS2Dynamiel, instead of sticking it (use 2.5*8mm screws).



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

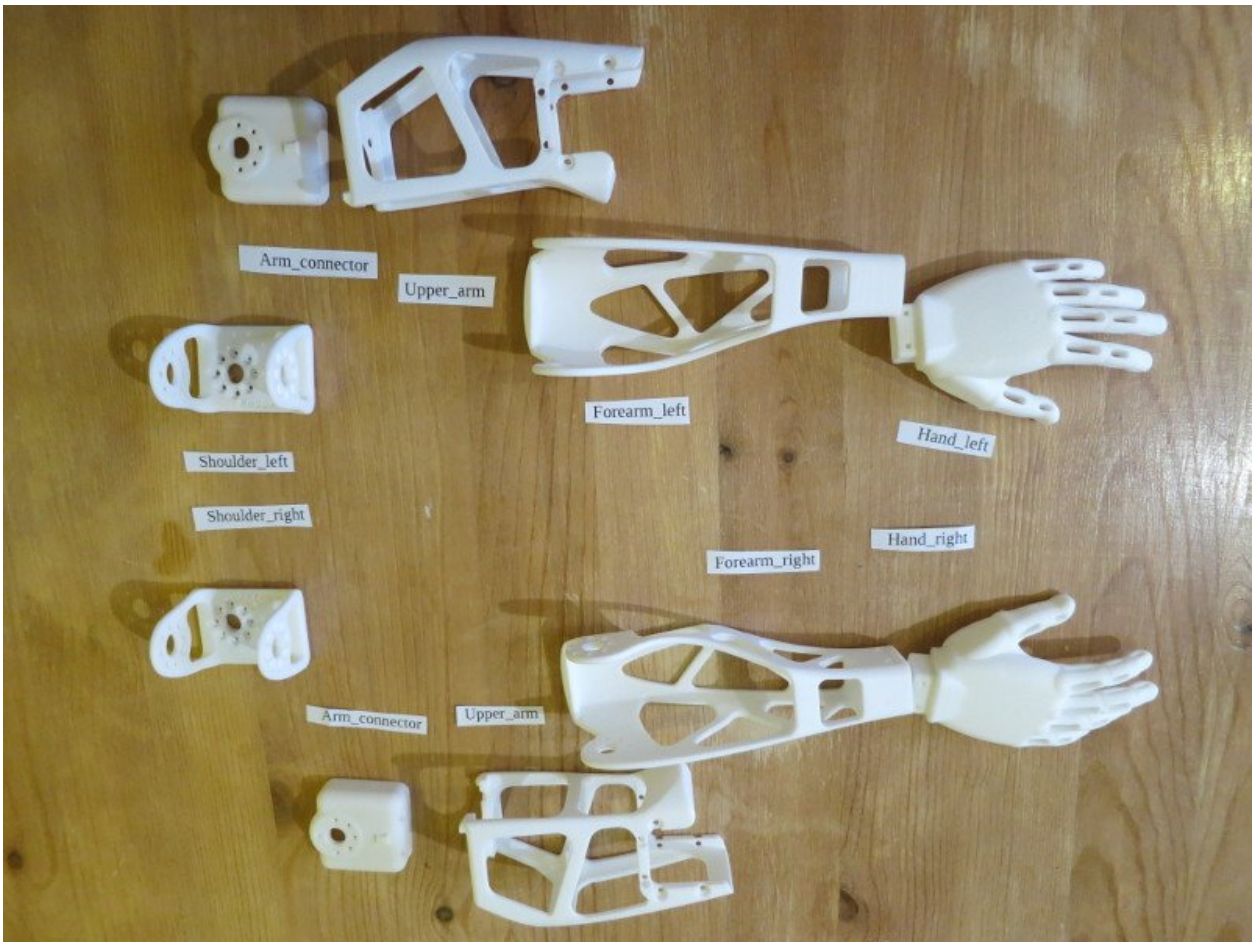
<< Back to menu

Arms assembly >>

Legs assembly >>

Head assembly >>

2.3.5 Arms assembly



\

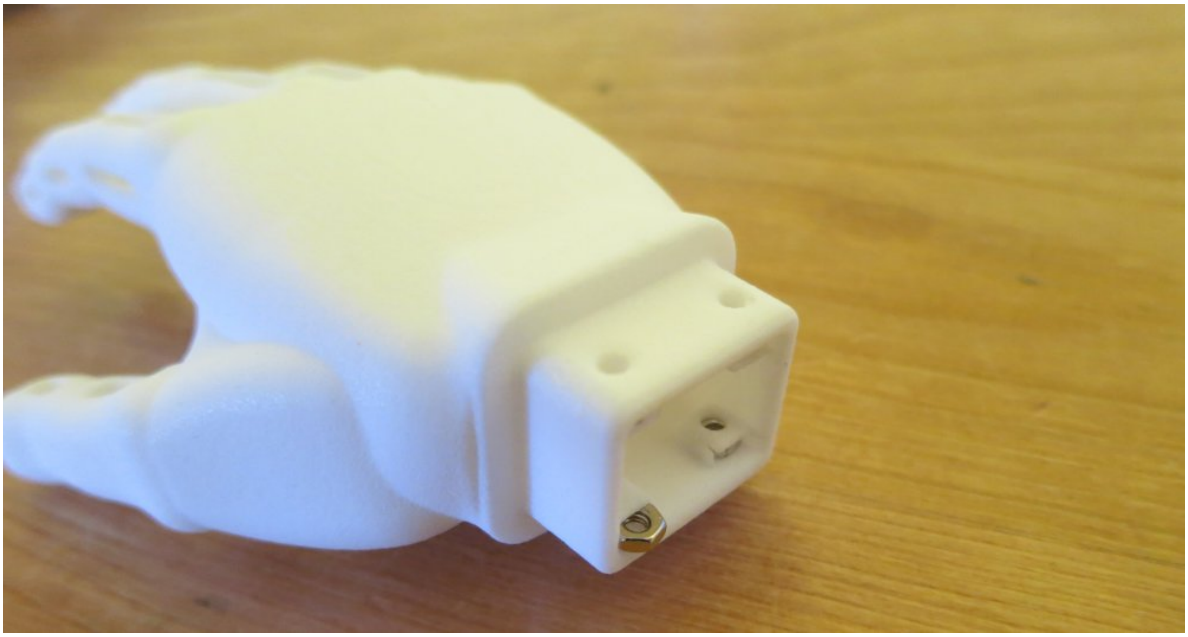
Motors lists:

| 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).

<< Back to menu

Trunk assembly >>

2.3.6 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.

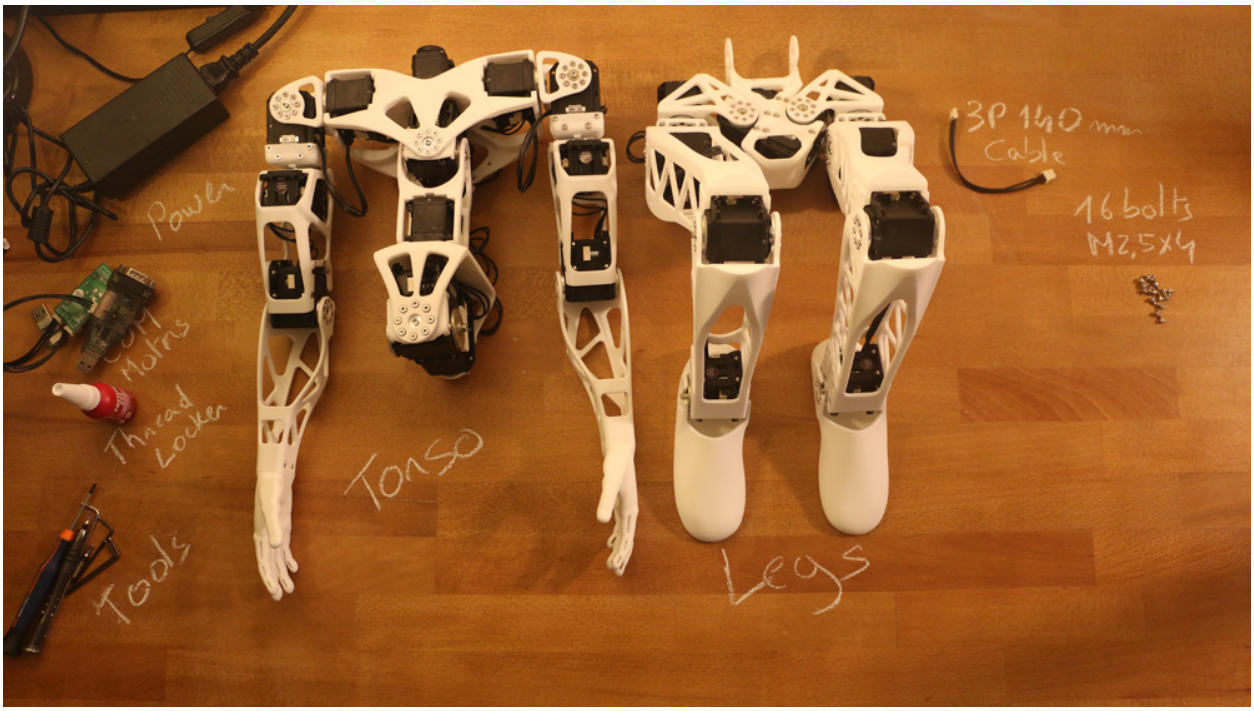
Right leg final assembly >>

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

<< Back to menu

Trunk assembly >>

2.4 Assembly guide for a Poppy Ergo Jr robot

COMMING SOON

CHAPTER

THREE

QUICKSTARTS

3.1 Discover your Poppy robot

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

3.1.1 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.

3.1.2 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')
```

3.1.3 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:
    name = m.name
    pos = m.present_position
    print "motor ",name," in position ',pos
```

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:
     name = m.name
    pos = m.present_position
    print "motor ",name," in position ',pos
```

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 non-compliant
poppy.compliant = True
poppy.head.compliant = False

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.

3.1.4 Sensors

TODO

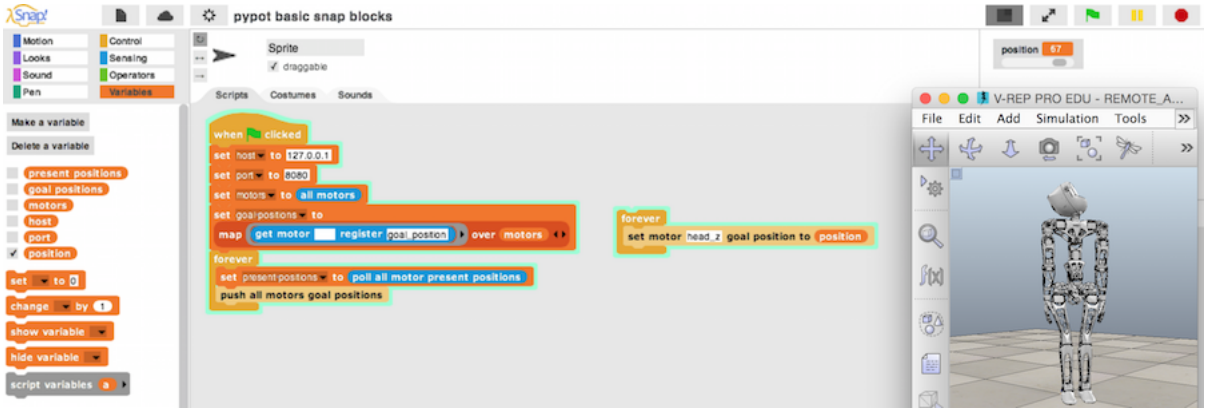
Next step is to use primitives, which is decribed in *this tuto*

3.2 Control a Poppy robot using Snap!

3.2.1 Introducing Snap!

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 .

3.2.2 Create the Snap! server

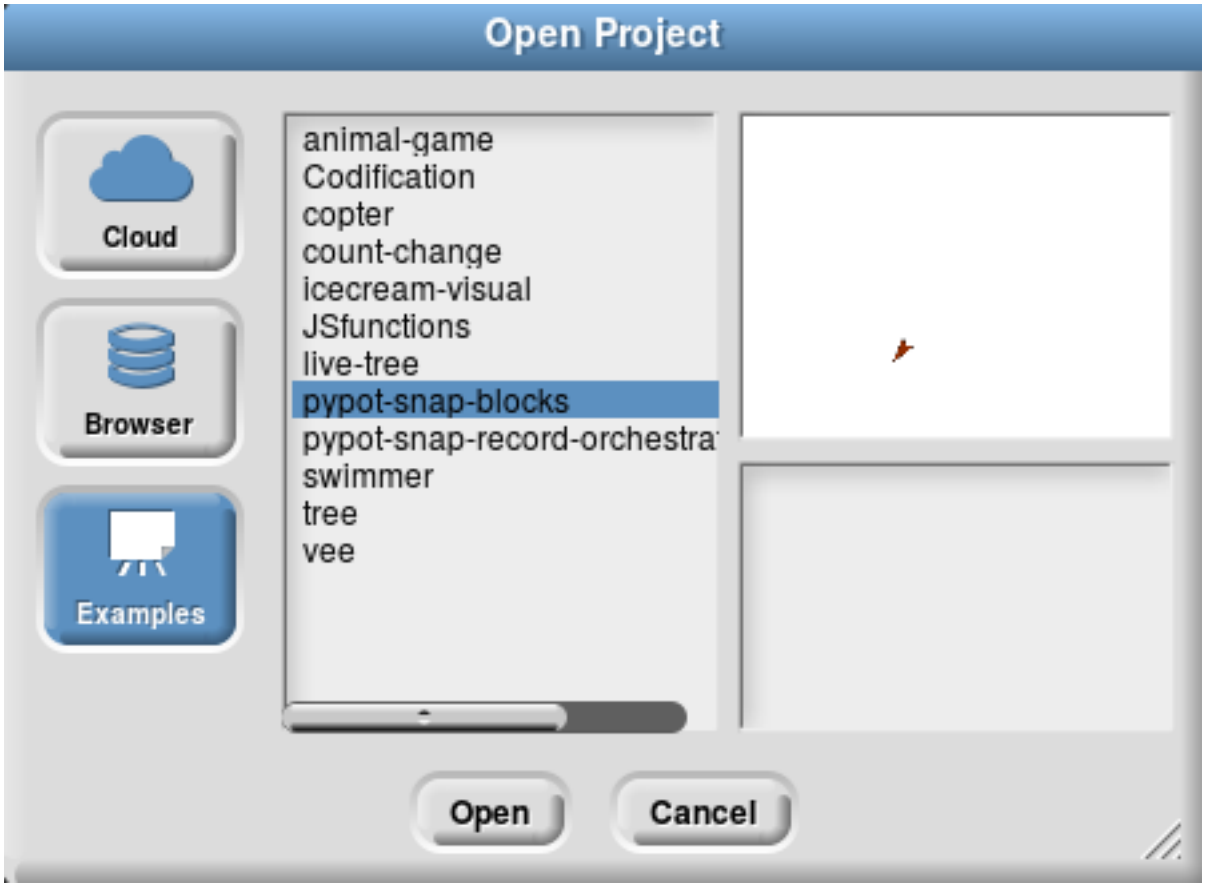
There are several ways of creating a Snap! server to control your robot.

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). 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:

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 .

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.

3.2.3 Controlling the robot

Pypot Snap! blocks

The Poppy Snap! blocks are the following:

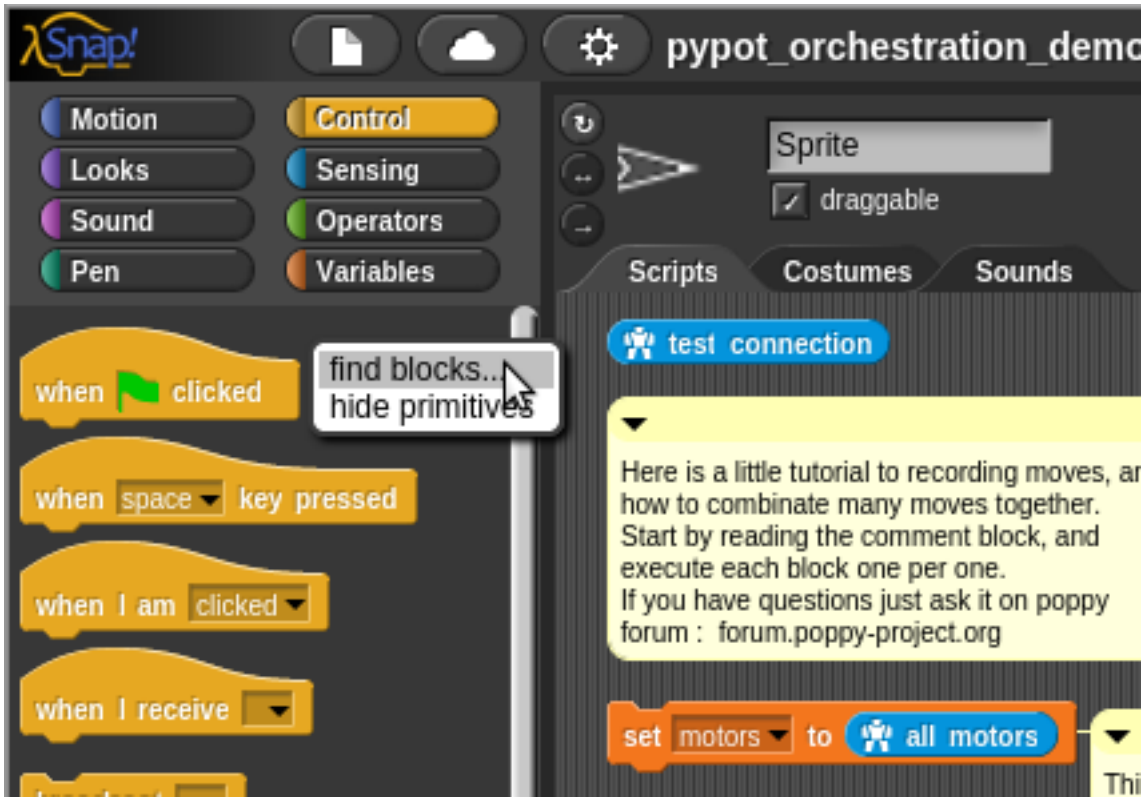

test connection
all motors
motors from alias
get motor motor_name register
get motor motor_name present position
motor index
all aliases
all primitives
all records
motor(s) goto position 0 in 0 seconds
set motor(s) compliant
set motor(s) register to
create move move name with motor(s)
stop record move 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

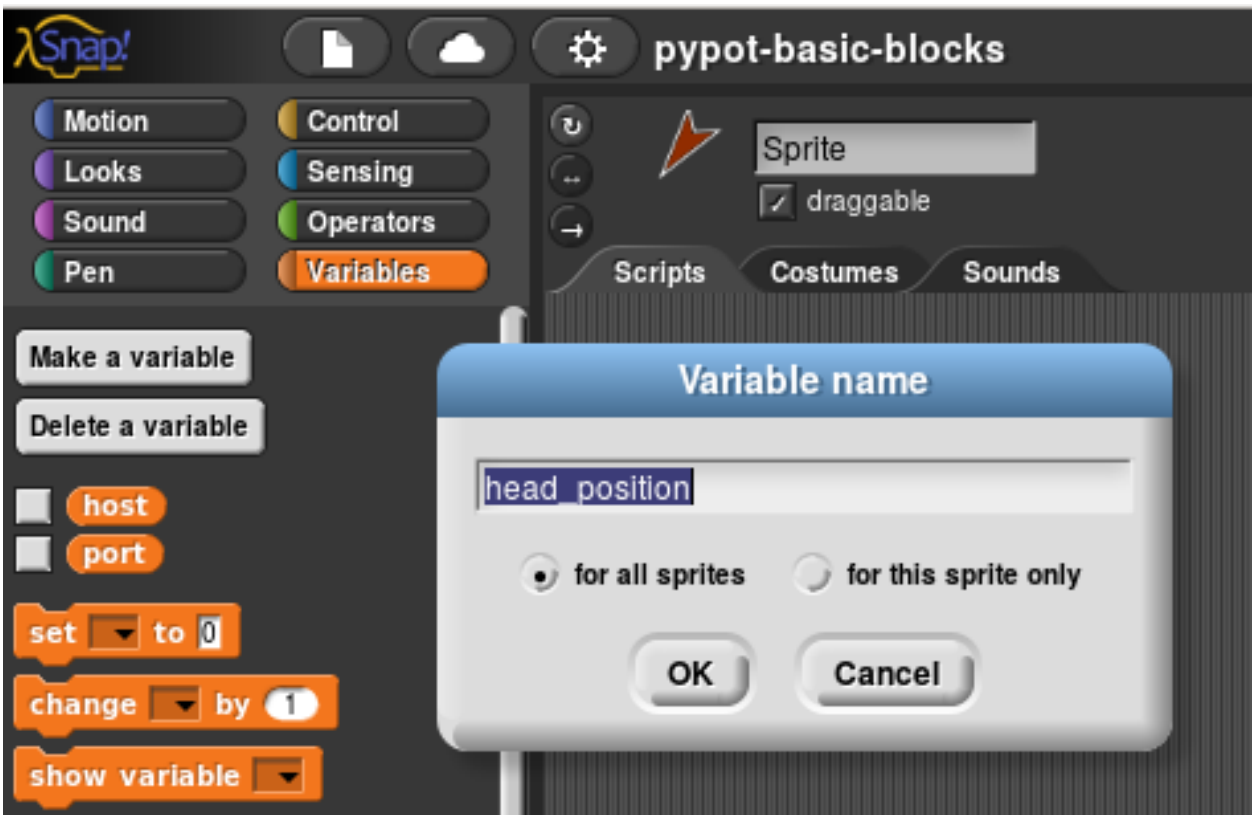
- set a motor position in a specified time
- turn a motor compliant or not
- set a register of a motor (e.g. set motor “head_z” register “present_load” to 10)
- create/attach a move to some motors (you have to create a move before to record or replay it)
- stop the record of a move
- start the record 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.



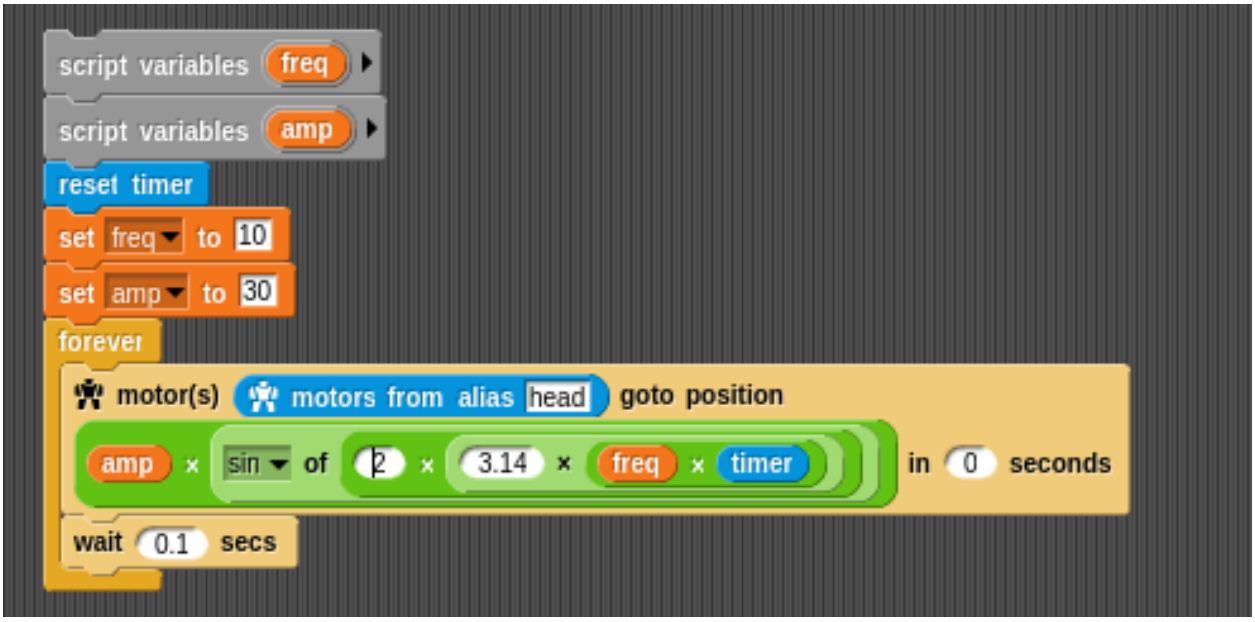
Then right click on it and use the slider option. Change the slider min/max to (-50, 50).

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.

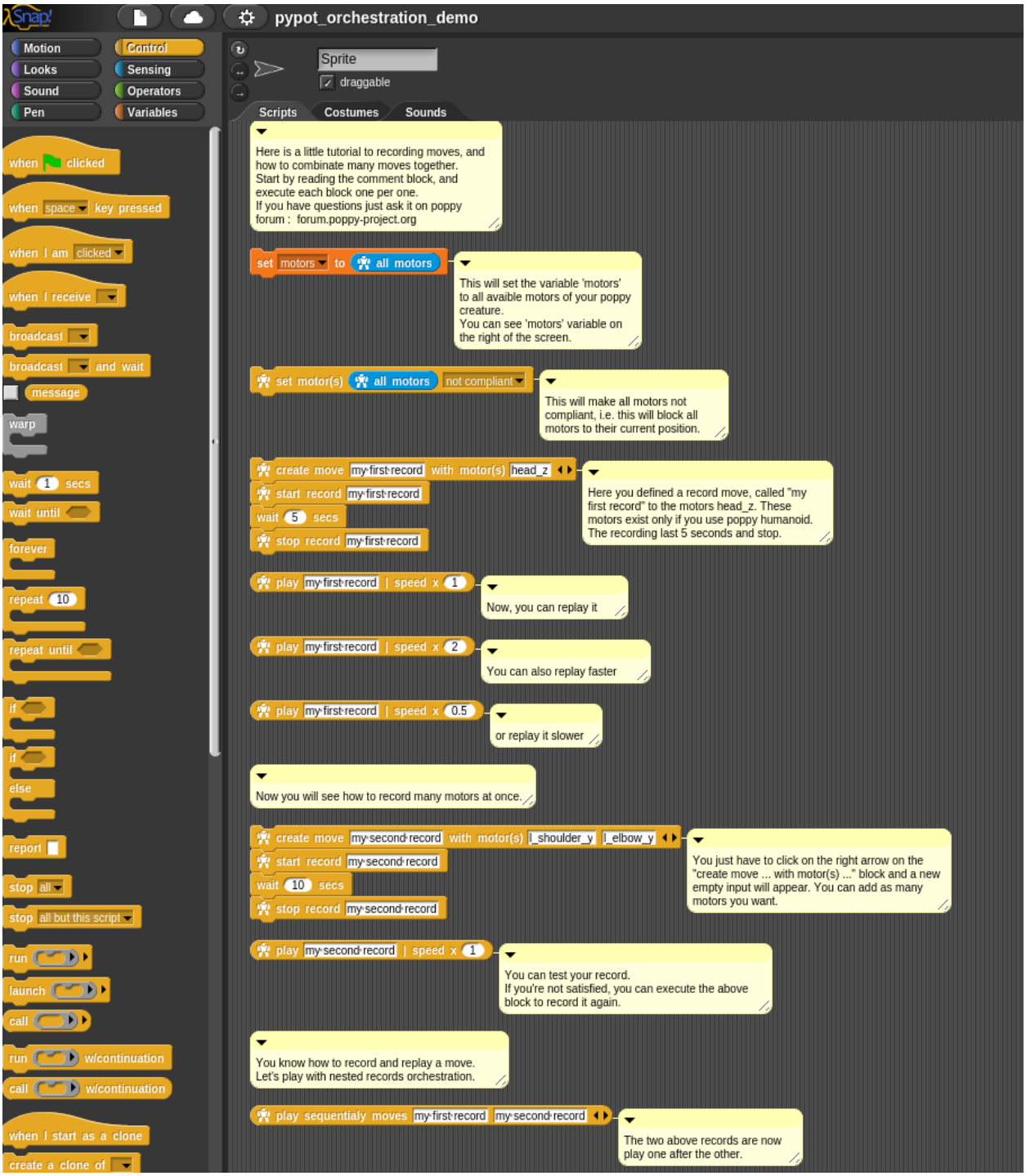
Can you, from this image, program your robot to say 'yes' or 'no' with the head?



3.2.4 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



3.3 Using V-rep to simulate a Poppy Humanoid robot

3.3.1 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. Or .

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.

3.3.2 Start a simulation with Poppy Humanoid

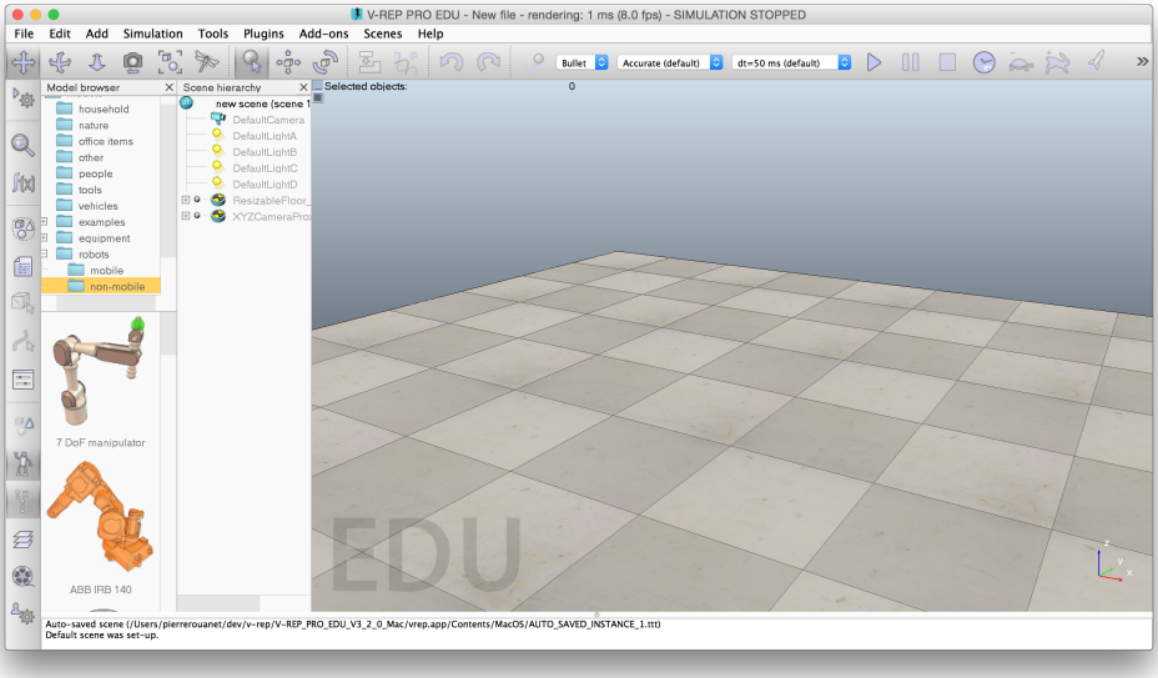
Once you have V-rep and poppy-humanoid installed on your computer (see here for installing poppy-humanoid and therefore pypot on your desktop computer), you need to follow these two steps:

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.

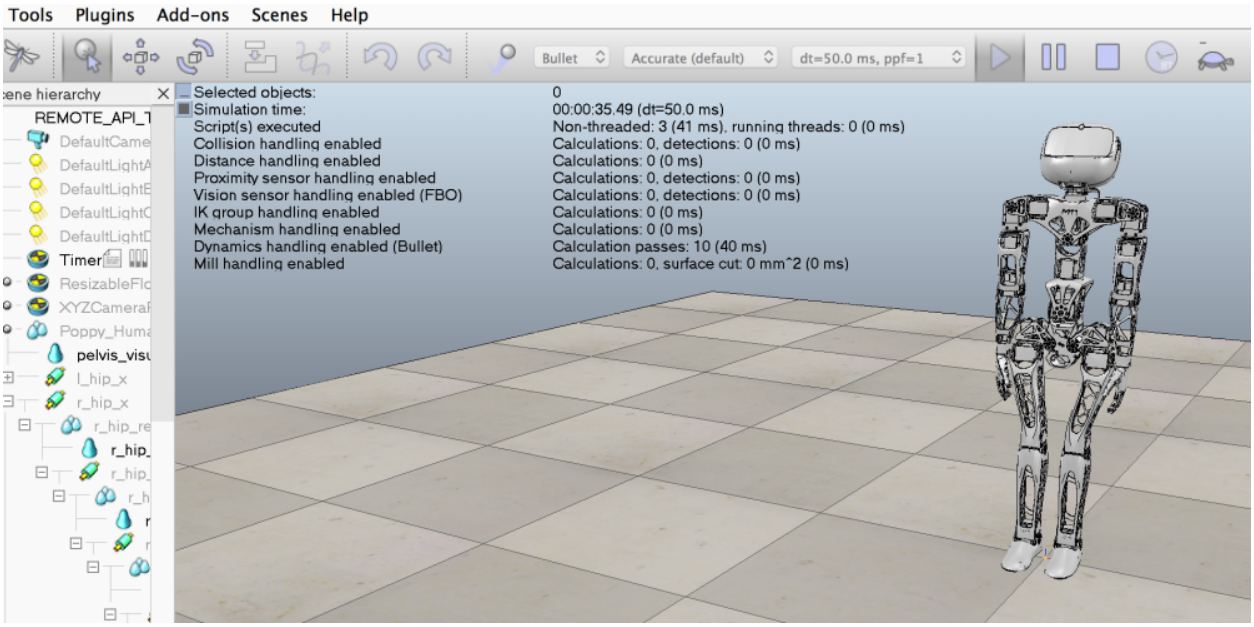
Launch a scene containing a Poppy Humanoid robot

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()
```

3.3.3 Troubleshooting and advanced usage

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

TODO

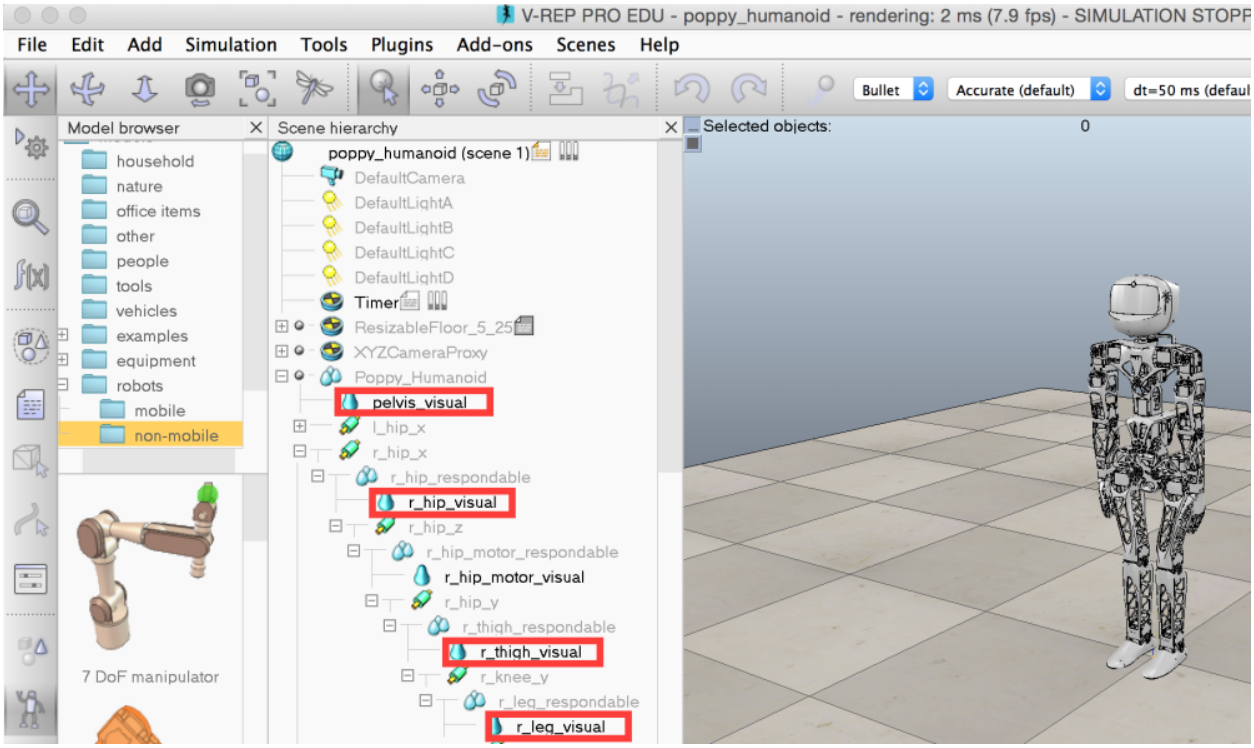
- what if it doesn't connect ?

- 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')
```

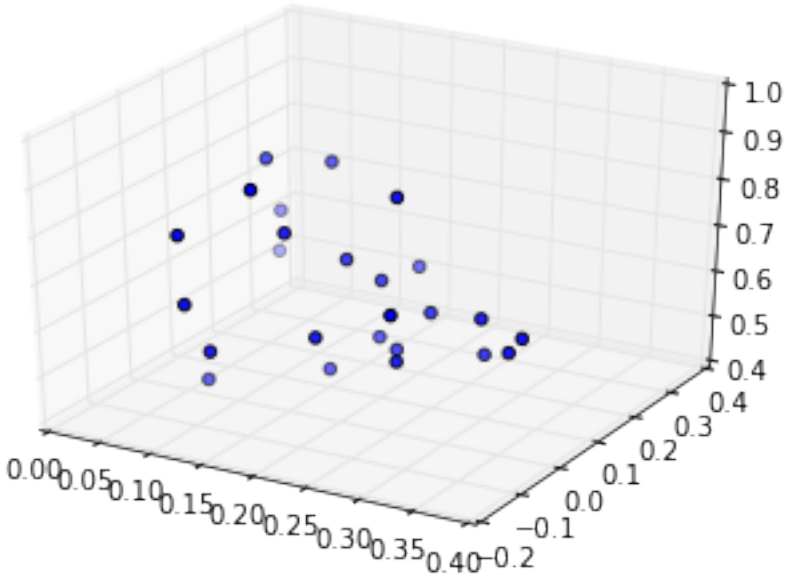
To discover the reachable space of the left hand of the robot (with respect to its head), you can for example generate many random positions for the arm (here, 25) and store the reached positions:

```
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_arm}
    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

FOUR

DEVELOPMENT GUIDES

4.1 Poppy-humanoid library

4.1.1 Introduction

Poppy Humanoid is the library allowing you to create a `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
```

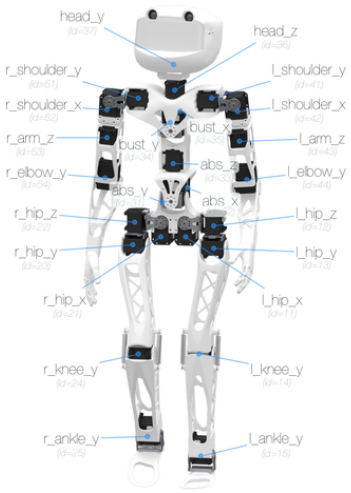
If you didn't change the default configuration file, this should connect to the 25 Dynamixel servomotors through two USB2AX.

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

4.1.2 Poppy Humanoid robot overview

TODO

Robot's motors list



4.1.3 Poppy Humanoid specific features

`PoppyHumanoid` is not only a `AbstractPoppyCreature` (which contains a `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 `StandPosition` named `stand_position`: the robot moves to standing position (0 to all motors).
- A `SitPosition` named `sit_position`: the robot moves to sitting position (legs bent with heels under the buttock)
- A `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 `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 `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 `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 `HeadIdleMotion` named `dance_beat_motion`: Slow and small movements of the head to have the robot look less 'dead'. This primitive is looping.
- A `ArmsTurnCompliant` named `arms_turn_compliant`: Automatically turns the arms compliant when a force is applied. This primitive is looping.
- A `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!

4.1.4 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*.

4.2 Poppy-torso library

4.2.1 Introduction

Poppy Torso is the library allowing you to create a `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.

4.2.2 Poppy Torso robot overview

TODO

image motors

list motors, move, etc

4.2.3 Poppy Torso specific features

`PoppyTorso` is not only a `AbstractPoppyCreature` (which contains a `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 `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 `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 `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 `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 `HeadIdleMotion` named `dance_beat_motion`: Slow and small movements of the head to have the robot look less 'dead'. This primitive is looping.
- A `ArmsTurnCompliant` named `arms_turn_compliant`: Automatically turns the arms compliant when a force is applied. This primitive is looping.
- A `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!

4.2.4 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*.

4.3 Poppy-ergo-jr library

4.3.1 Introduction

Poppy Ergo Jr is the library allowing you to create a `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*.

4.3.2 Poppy Ergo Jr robot overview

TODO

image motors

list motors, move, etc

4.3.3 Poppy Ergo Jr specific features

`PoppyErgoJr` is not only a `AbstractPoppyCreature` (which contains a `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 `BasePosture` named `base_posture`: the robot goes in an initial posture.

- A `RestPosture` named `rest_posture`: the robot goes in a 'resting' posture.
- A `Dance` named `dance`: Simple dance based on sinusoidal motions. This primitive is looping.
- A `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 `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!

4.3.4 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*.

4.4 Poppy-creature library

4.4.1 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 `AbstractPoppyCreature` which takes a configuration and builds a `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.

4.4.2 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)

4.4.3 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 `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 `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 give it to the community! Most interesting creatures will be added to this documentation!

4.4.4 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):

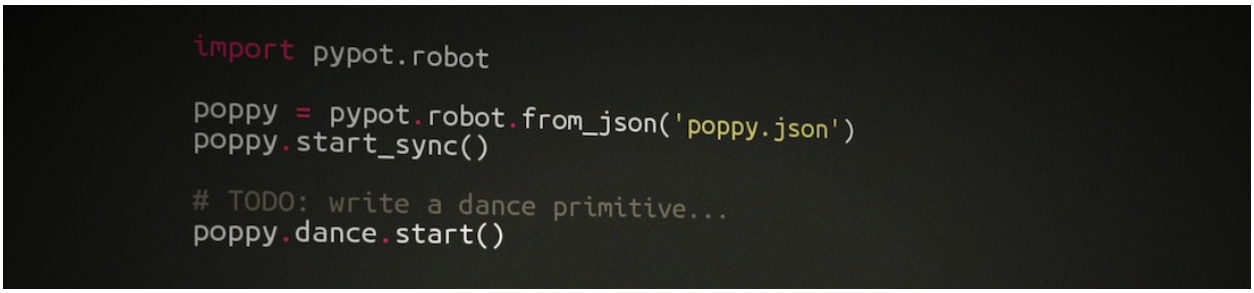
```
python setup.py install
```

The requirements for Poppy Creature are *Pypot* and bottle.

4.5 Pypot library

4.5.1 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.

The sources are available on Github.

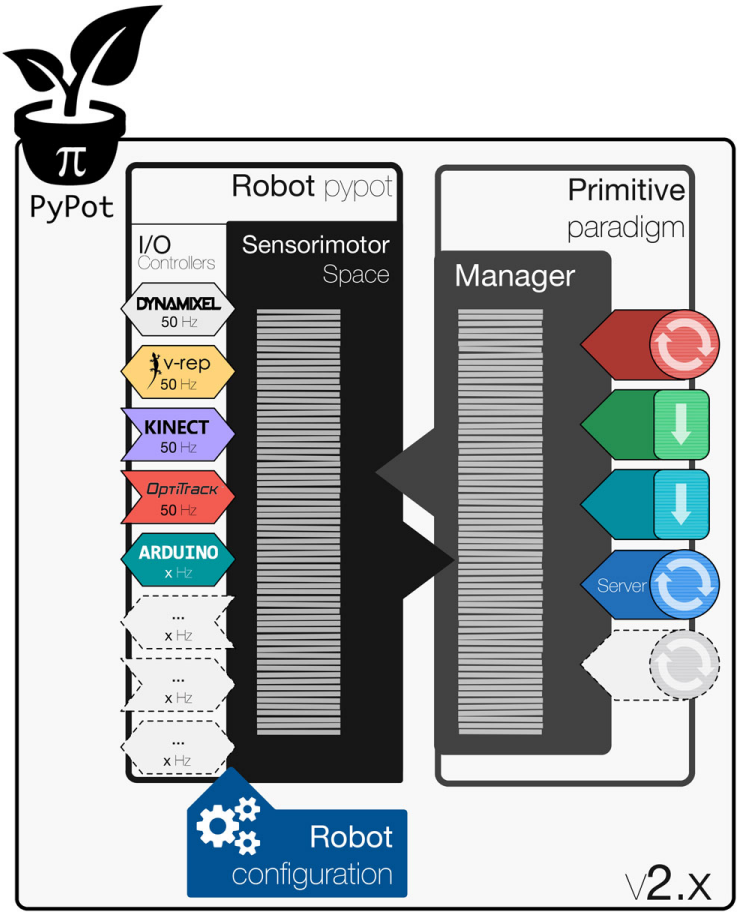
Pypot's architecture

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 next section when dynamixel IO is replaced by the communication layout with the VREP simulator.

- **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 `Controller` is the `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 `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 `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 `AbstractController`.

The IO should also handle the low-level communication errors. For instance, the `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 `DxlIO` and the `VrepIO` ones.

Writing a new Controller

A `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 `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 `DxlMotor`. It also reads the goal position/speed/load set in the `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 `AbstractController`, which does exactly that. If your controller fits within this conception, you should only have to overide the `update()` method.

In the case of the `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 `from_config()` or `from_vrep()`) allowing users to easily instantiate their `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 `DxlController` and the `VrepController` only interact with the `Robot` through getting and setting values into `Motor` instances, they can be directly switch.

4.5.2 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 *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 `move` module contains utility classes to help you record and play moves. A `Move` object simply contains a sequence of positions.

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

Assuming you have a `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 second.
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*.

4.5.3 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 `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 `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 `dynamixel` module offers several useful function to learn the state of your connections:

- `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.
- `find_port` takes a list of motors IDs and scan the available ports until it finds the motors.
- `autodetect_robot` scans all ports and creates a `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 `close()` method:

```
dxl_io.close()
```

Note: The class `DxlIO` can also be used as a Context Manager (the `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 `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 XL-320\n",
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_baudrate

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_baudrate
```

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 `Robot` you can:

- automatically initialize all connections (make transparent the use of multiple USB2serial connections),
- define `offset` and `direct` attributes for motors,
- automatically define accessor for motors and their most frequently used registers (such as `goal_position`, `present_speed`, `present_load`, `pid`, `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 _quickstart-configfile 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**
   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 `config`, you can find the configuration dictionnary of a Poppy Ergo Jr robot.

Use the configuration To create a `Robot` object from a Python dictionnary, use the `from_config()` function function:

```
import pypot.robot

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

You can also create a `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 `Robot`.

Dynamixel controller and Synchronization Loop

The `Robot` held instances of `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 `Robot` class uses a `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 `Robot` uses a particular controller `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 `start_sync()`. You can also stop the synchronization if needed (see the `stop_sync()` method).

Note: With the current version of pypot, you can not yet indicate in the configuration which subclasses of `DxlController` you want to use. If you want to use your own controller, you should either modify the config parser, modify the `BaseDxlController` class or directly instantiate the `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 `Robot` are:

- motors: list of `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 `Robot` class:

- `power_up()` to set maximum torque and remove compliance.

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

This is especially useful for choregraphies, because the `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 `Robot`, you should always call the `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 `close()` method on a `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 my_prim 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 `close()` method to terminate (please refer to start_prim for details).

The Motor object

Overview

The `Robot` class contains a list of `DxlMotor`, each `DxlMotor` being linked to a physical Dynamixel motor.

Registers

This class provides access to (see `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
    print "---"
```

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 teh 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 `lower_limit` and `upper_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 `goal_speed` attribute to the desired value in degree per seconds. This automatically sets the `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 `moving_speed`. Nevertheless, while the motor will turn infinitely with the wheel mode, here with the `goal_speed` the motor will still respect the angle limits.

Warning: If you set both `goal_speed` and `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 `goto_position()` function If you want a servo to go to a certain position in a certain time (for synchronization reasons...), use the `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 `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 `Robot` abstraction. `Primitive` allows you to create more complexe, parallelized behavior really easily.

What is a "Primitive"?

We call `Primitive` any simple or complex behavior applied to a `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 `start()`, `stop()`, `pause()` and `resume()`. Unlike regular python thread, primitive can be restart by calling again the `start()` method.

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

The `PrimitiveLoop` is a `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 `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 `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 `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:

- `Cosinus` for a cosinus move
- `Square` for a square (go to max position, wait duty*cycle time, go to minposition, wait (1 - duty)*cycle time)
- `PositionWatcher`: records and saves all positions of the given motors. You have a plot function to see your data.
- `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 `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 `Robot` (resp. `DxlMotor`) instance is transformed into a `MockupRobot` (resp. `MockupMotor`). Those class are used to intercept the orders sent and forward them to the `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 `DxlMotor` into `MockupMotor` thanks to the `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 `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 `should_stop()` and `should_pause()` in your run loop. You can also use the `wait_to_stop()` and `wait_to_resume()` to wait until the commands have really been executed.

Note: You can refer to the source code of the `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 `__init__()` method.

Note: You should always call the super constructor if you override the `__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 `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 constructor
    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.

4.5.4 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 `VrepIO` designed to communicate with V-REP through its remote API.

This IO can be used to:

- connect to the V-REP server : `VrepIO`
- load a scene : `load_scene()`
- start/stop/restart a simulation : `start_simulation()`, `stop_simulation()`, `restart_simulation()`
- pause/resume the simulation : `pause_simulation()`, `resume_simulation()`
- get/set a motor position : `get_motor_position()`, `set_motor_position()`
- get an object position/orientation : `get_object_position()`, `get_object_orientation()`

Connecting to V-REP

First launch V-rep.

Then create your `Robot` using the `from_vrep()` method, providing the vrep host, port and scene instead of the `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 `Robot` returned will also provide a convenience 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 `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 `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 `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 vrap-port arguments:

```
from poppy_humanoid import PoppyHumanoid

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

REST API

Any `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 `HTTPRobotServer`)
- ZMQ socket (see the `ZMQRobotServer`)

The `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: `snap` 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 and the port

#how the read a register
req = {"robot": {"get_register_value": {"motor": "head_z", "register": "present_position"}}}
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_position", "value": "40"
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 (see section log_ex).

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 `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 `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.
- `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 `robot` is logging when the synchronization is started/stopped (INFO) and when a primitive is attached (INFO).
- `primitive` logs a message when the primitive is started/stopped and paused/resumed (INFO). Each `update()` of a LoopPrimitive is also logged (DEBUG). Each time a primitive sets a value to a register a message is also logged (DEBUG).
- `manager` provides you information on how the values sent within primitives were combined (DEBUG).
- `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.

Warning: Herborist is entirely written in Python but requires PyQt4 to run.

More precisely, Herborist can be used to:

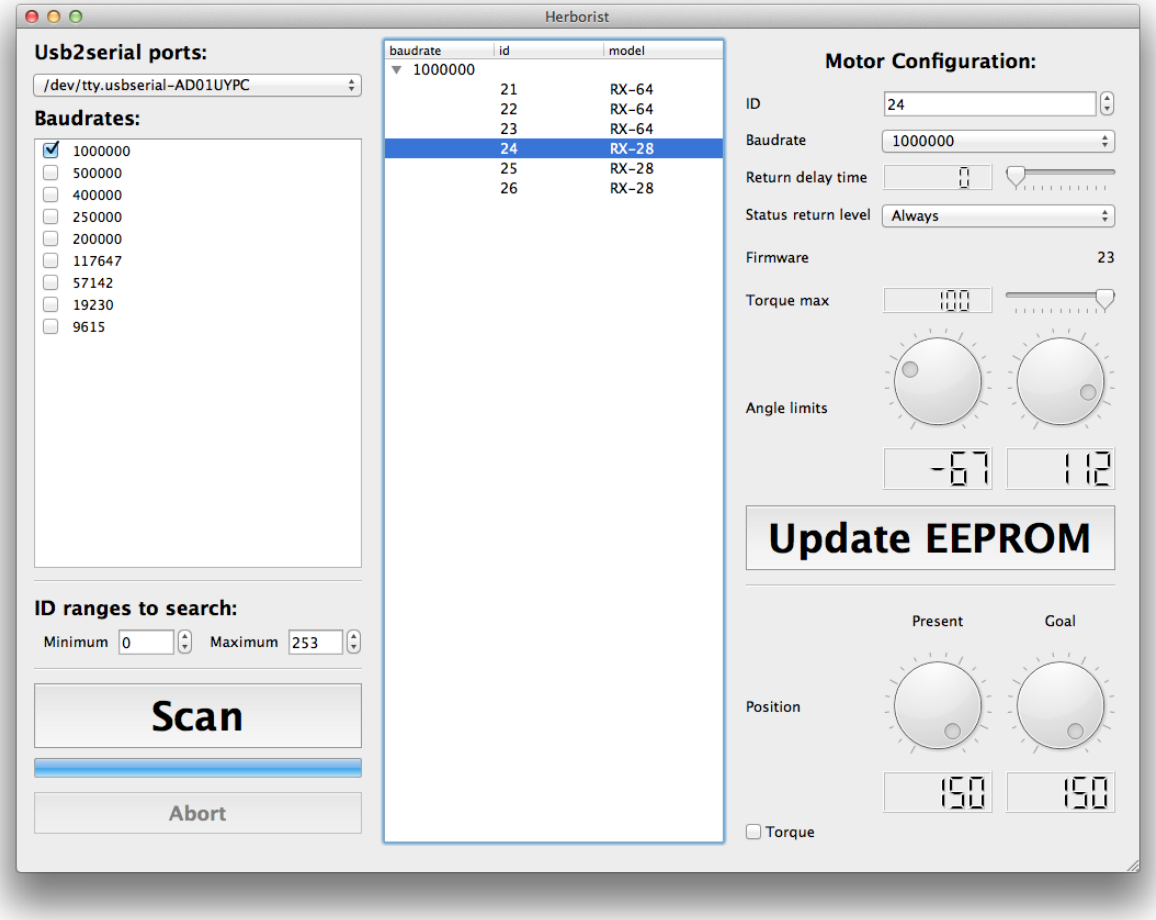
- Find and identify available serial ports
- Scan multiple baud rates to find all connected motors
- Modify the EEPROM configuration (of single or multiple motors)
- Make motors move (e.g. to test the angle limits).

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.

4.6 APIs

4.6.1 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

4.6.2 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

4.6.3 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

4.6.4 pypot package

Subpackages

pypot.dynamixel package

Subpackages

pypot.dynamixel.io package

Submodules

pypot.dynamixel.io.abstract_io module

pypot.dynamixel.io.io module

pypot.dynamixel.io.io_320 module

Module contents

pypot.dynamixel.protocol package

Submodules

pypot.dynamixel.protocol.v1 module

pypot.dynamixel.protocol.v2 module

Module contents

Submodules

pypot.dynamixel.controller module

pypot.dynamixel.conversion module

pypot.dynamixel.error module

pypot.dynamixel.motor module

pypot.dynamixel.syncloop module

Module contents

pypot.primitive package

Submodules

pypot.primitive.manager module

pypot.primitive.move module

pypot.primitive.primitive module

pypot.primitive.utils module

Module contents

pypot.robot package

Submodules

pypot.robot.config module

pypot.robot.controller module

pypot.robot.io module

pypot.robot.motor module

pypot.robot.remote module

pypot.robot.robot module

pypot.robot.sensor module

Module contents

pypot.sensor package

Subpackages

pypot.sensor.camera package

Submodules

pypot.sensor.camera.abstractcam module

pypot.sensor.camera.opencvcam module

pypot.sensor.camera.rpicam module

Module contents

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

Module contents

Submodules

pypot.sensor.optibridge module

pypot.sensor.optitrack module

Module contents

pypot.server package

Submodules

pypot.server.httpserver module

pypot.server.rest module

pypot.server.server module

pypot.server.snap module

pypot.server.zmqserver module

Module contents

pypot.tools package

Subpackages

pypot.tools.herborist package

Submodules

pypot.tools.herborist.herborist module

Module contents

Submodules

pypot.tools.dxl_reset module

Module contents

pypot.utils package

Submodules

pypot.utils.appdirs module

pypot.utils.interpolation module

pypot.utils.pypot_time module

pypot.utils.stoppablethread module

pypot.utils.trajectory module

Module contents

pypot.vrep package

Subpackages

pypot.vrep.remoteApiBindings package

Submodules

pypot.vrep.remoteApiBindings.vrep module

pypot.vrep.remoteApiBindings.vrepConst module

Module contents

Submodules

pypot.vrep.controller module

pypot.vrep.io module

Module contents

Submodules

pypot.kinematics module

Module contents

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search