

Poppy

Documentation

release 1.1.0



PyPot

Table of Contents

Introduction	0
Getting started	1
Build a robot	1.1
Connect to the robot	1.2
Program the robot	1.3
Visualize	1.4
Examples of projects	1.5
Startup	2
Write an ISO on Windows	2.1
Write an ISO on Mac OSX	2.2
Write an ISO on GNU/Linux	2.3
Assembly guides	3
Assemble the Ergo Jr	3.1
Electronic assembly	3.1.1
Motor configuration	3.1.2
Hardware construction	3.1.3
Assemble the Poppy Humanoid	3.2
Assemble the Poppy Torso	3.3
Programming	4
Programming with Snap!	4.1
Usage of Jupyter notebooks	4.2
Programming in Python	4.3
Robots APIs	4.4
Activities	5
Link with Snap4Arduino	5.1
Installation for advanced users	6
From simulation to real robot	7
Snap! on the real robot	7.1
Program with Jupyter notebooks on the real robot	7.2
Software libraries	8
Pypot	8.1
Poppy-creature	8.2
Poppy Ergo Jr	8.3
Poppy Humanoid	8.4
Poppy Torso	8.5
Appendix	9
Contribute	9.1
Raspoppy	9.2
Glossary	

0Introduction

0.1About

Welcome to the manual of the [Poppy Project](#), an open-source robotic platform.

In this documentation, we will try to cover everything from the short overview of what is possible within the project to the details on how you can build a Poppy robot or reproduce one of the pedagogical activity.

This book is licensed under the [Creative Commons Attribution-ShareAlike 4.0 license](#). The source can be found and modified on [GitHub](#).

In the first chapter, we will give you a simple but exhaustive overview of what you can do within this project, so you can quickly focus on the following chapters that cover what you are really interested in. While some advanced chapters may require a good understanding of mechanics, electronics or computer sciences, the [Getting Started](#) section is intended to be easily accessible by all readers.

0.2Version

This is the version 1.0 of the Poppy documentation, updated at Tue Apr 05 2016 01:41:06 GMT+0000 (UTC).

It contains the documentation for:

- Poppy Humanoid hardware version 1.0.1
- Poppy Torso hardware version 1.0.1
- Poppy ErgoJr hardware version beta 6
- 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 Getting Started



[Poppy Project](#) 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, who all share a vision: robots are powerful tools to learn and be creative.

The [Poppy community](#) develops robotic creations that are easy to build, customize, deploy, and share. We promote open-source by sharing hardware, software, and web tools.

The Poppy project has been originally released by [Inria Flowers](#).

This project aims at developing a robotic open-source and open-hardware kits, for the conception and programming of robotic creatures. These kits also contain an integrated software environment, easy to install, dedicated to the use of the robot, and associated with a web platform enabling the community to share experiences and to contribute to its improvement.

Recently, Poppy robots have been used as a educational and motivational tool to learn engineering and digital sciences.

The Poppy project also consists of a very active and diverse community. People of different horizons collaborate to improve the project; they add features to the core libraries, release advanced behaviors, create pedagogical contents, design artistic performances, or even new robots.

To ease these exchanges two supports are available:

- [The poppy-project forum](#) for help, discussions and sharing ideas.
- [GitHub](#) to submit your contributions.

All sources of the Poppy Project (software and hardware) are available on [GitHub](#).

1.1 The Poppy creatures

Poppy creatures are open-source robots, available for download and modification ([Creative Commons Attribution-ShareAlike](#) for the hardware and [GPLv3](#) for software). They were designed with the same principles in mind.

All Poppy creatures:

- are made from pieces of printable 3D and Dynamixel motors,
- use an embedded board for control (a Raspberry Pi 2 or Odroid for older versions),
- are based on a Python library, [pypot](#), allowing to control Dynamixel servomotors in an easy way,
- have a simulated version available (based on V-REP),
- can be controlled using a visual programming language ([Snap!](#) a variation of Scratch) and a textual language [Python](#). They are also programmable through a REST API, which enables the control with other programming language (connection for Matlab and Processing already exists) - see Chapter #TODO for details,
- come with associated documentation, tutorials, examples, pedagogical activities.

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

You can buy a full Poppy Creature robot from our [official retailer](#).

You can also get all the parts yourself by following the bill of materials (see below).

1.1.1Poppy Ergo Jr

The Poppy Ergo Jr robot is a small and low cost 6-degrees-of-freedom robot arm. It is made of 6 cheap motors (XL-320 Dynamixel servos) with 3D-printed parts simple to design.

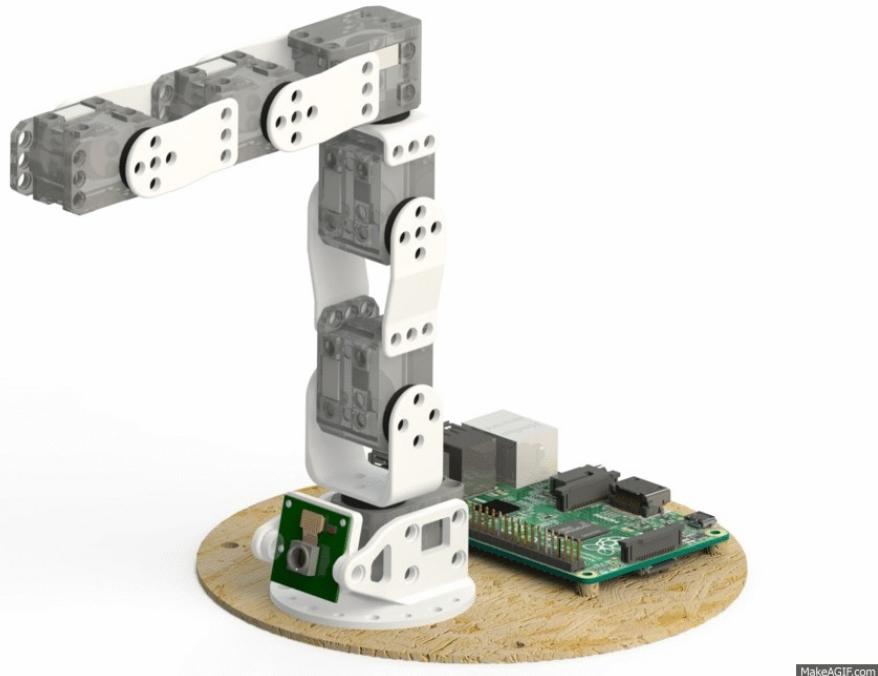


The 3D parts were made so they can be easily printed on a basic 3D printer and the motors are only 20 € each. Its electronic card access is simple. It makes it easy to connect extra sensors and is perfect for pedagogical purposes.

You can choose among several tools to use for the end of its arm:

- A lamp.
- A gripper.
- A pen holder...

Rivets make them easy to change, depending on the type of activities you are planning.



The Ergo Poppy Jr is ideal to start manipulating robots and learn robotic without difficulties.

Its low price, simple assembly, and easy control make it suited for educational purpose.

You can get all the parts yourself following the [bill of materials](#) (BOM) and printing the [3D files](#) available in the STL format.

For more informations, check the [assembly guide of the Ergo Jr](#)

1.1.2Poppy Humanoid

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

You can get all the parts yourself following the [bill of materials](#) (BOM) and printing the [3D files](#) available as STL, STEP and Solidworks 2014 format.



1.1.3 Poppy Torso

It is the upper part of Poppy Humanoid (13 degrees of freedom). Poppy Torso is more affordable than the complete Poppy Humanoid, which makes it more suitable to uses in an educational, associative and makers context. Poppy Torso can be a good medium to learn science, technology, engineering and mathematics (STEM).

You can get all the parts yourself following the [bill of material](#). The [3D models](#) for the parts are the same as Poppy Humanoid, without the legs and with an extra [suction cup support](#).



1.1.4 Other interesting Poppy Creatures

A key aspect of the Poppy Project is to stimulate creativity and experimentation around robotics. We try to provide all the tools needed to design new robots based on the same technological bricks. Some new creatures are in development within the community. Some of them are presented in this section below.

1.1.4.1 Poppy right arm (work in progress)

Poppy right arm is a Poppy creature based on a right arm of Poppy Humanoid, with 3 additional XL-320 motors at the end to improve the reach and agility of the arm. It used the same gripper tool used in the Ergo Jr, designed to grab simple objects.

The project was realized during an internship at Inria Flowers by [Joel Ortiz Sosa](#). Find more info and the sources in its [repository](#).

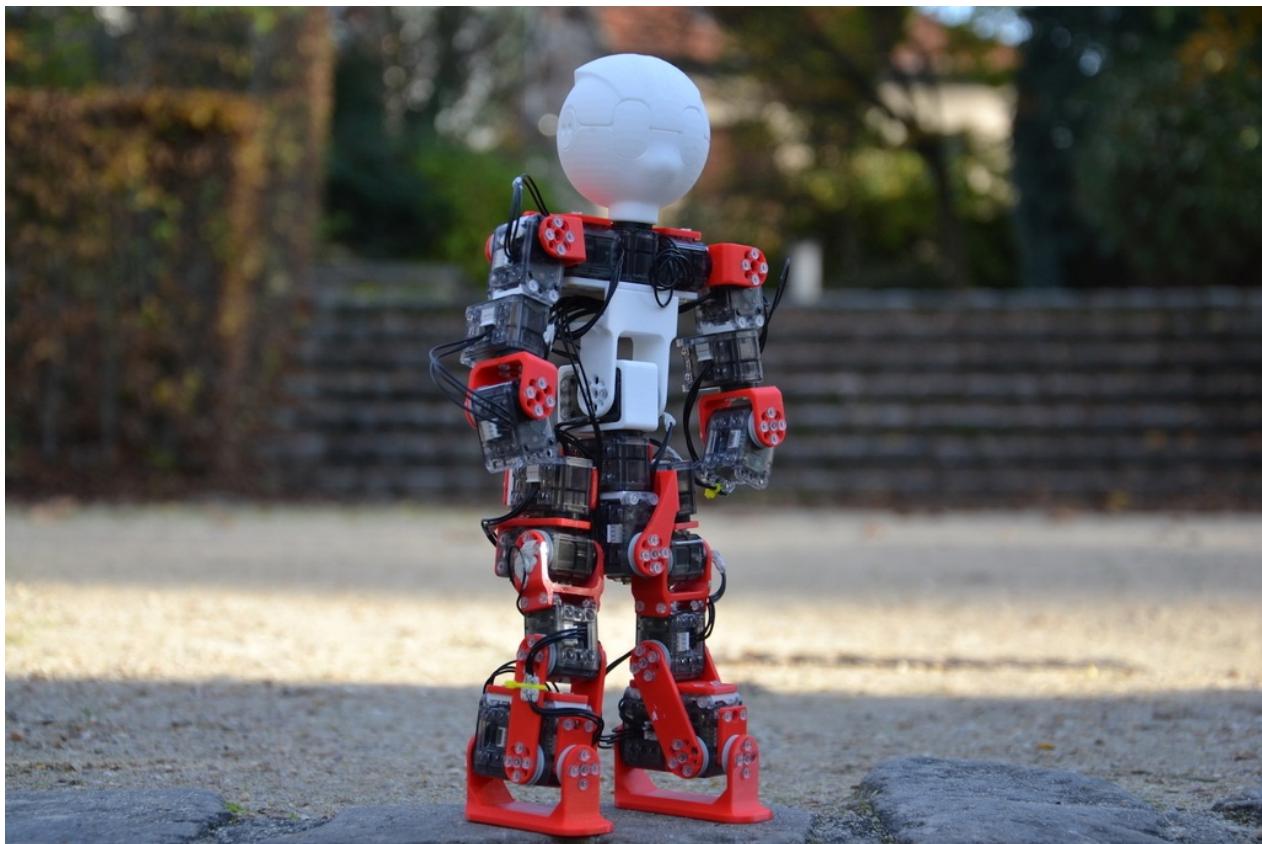
1.1.4.2 Small and low-cost humanoids

1.1.4.2.1 Heol

Heol - meaning "sun" in Breton - is a humanoid robot 34 cm tall made by association [Heol robotics](#). 23 motors compose it, all other parts are 3D printed. It also uses the Pypot library for its movements.

Heol's purpose is to put a smile on ill children's faces. It has the ability to be an educational tool by becoming a learning support materials such as programming and mechanical design.

Its participation in the RoboCup (World Cup football for robot) is also envisaged.



1.1.4.2.2Poppyrate



1.1Build the robot

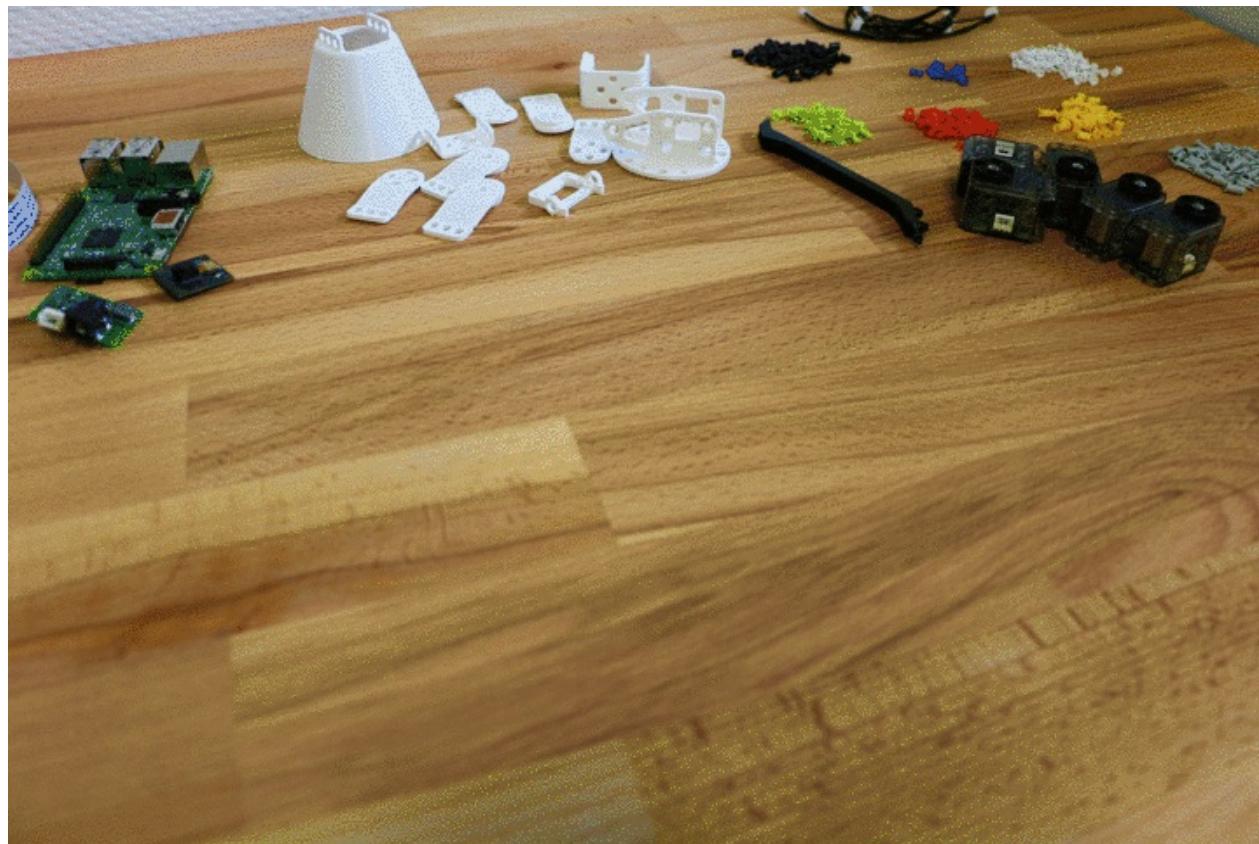


Depending on the Poppy robots you are planning to use, the assembly time, required skills, tools and difficulty may vary a lot. Building an Ergo Jr should not take more than one hour and no specific tool is needed while assembling an entire Poppy Humanoid may take a few days and quite a lot of screws!

This section intends to give you hints and an approach of some critical points so you aware of them before digging into the construction. We will also point to each dedicated chapter where you will find the resources and detailed step by step assembly procedure for each robot.

1.0.1Assembling an Ergo Jr

You can find a full assembly documentation in the chapter [step by step assembly of an Ergo Jr.](#)



The Ergo Jr robot was designed to be a simple little robot, cheap and easy to use. The 3D parts were made so they can be easily printed on a basic 3D printer and the motors (6 XL-320 Dynamixel servos) are only 20\$ each.

The Ergo Jr is very easy to build and its end effector can be easily changed - you can choose among several tools: a lamp, a gripper, a pen holder...

Thanks to OLLO rivets the robot is very simple to assemble. These rivets can be removed and added very quickly with the OLLO tool. It should not take more than one hour to entirely built it, which allows great design freedom.

Except from **checking the motor orientation**, there is not really any pitfall. If you are familiar with Lego bricks, you should be able to assemble an Ergo Jr without much problem! In any cases, the rivets are made to be as easy to assemble than to disassemble, so in case of problem you can just start over!

Also make sure, to [configure your motors](#) before assembling the robot as it is harder to do after!

1.0.2 Assembling a Torso or a Humanoid

You can find a full assembly documentation in the chapter [step by step assembly of a Poppy Humanoid](#).



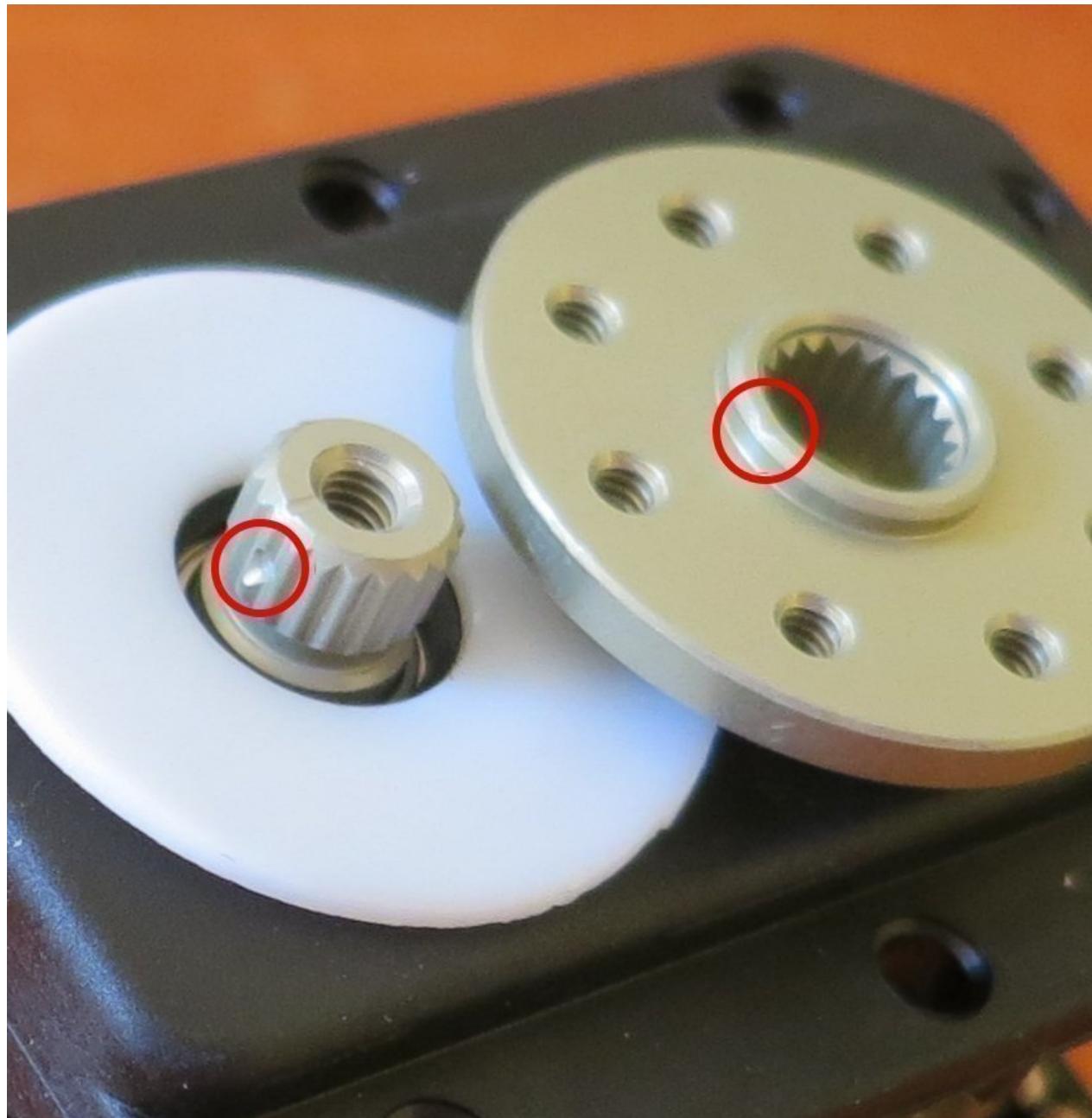
Building a Poppy Torso or a Humanoid is more complex than a Ergo Jr but it is not really more complicated than building a Meccano or some Swedish furniture. It mainly consists on those few steps:

- assemble the horn on each motors: **you will have to be really cautious about the motor zero position!**
- configure the motors so they match the *poppy configuration*
- use a lot of screws to connect all 3D printed parts to the motors
- do a bit of electronic for the embedded board inside the head: this can be a bit tricky if you are not familiar with electronics.

Patience and precision are your allies, but in case of errors do not panic: Poppy is a robot intended to be assembled and disassembled. If you pay attention to the few **warnings** bellow, and with a few trials and errors you will have a working Poppy Torso or POppy Humanoid:

Warning 1: The Poppy humanoid and torso robots are built using mainly MX-28 and MX-64 Dynamixel servomotors, which are pretty powerful and may be harmful to your fingers or materials. So be very careful and put the robot in a free space while testing it.

Warning 2: put the dot on the horn at the same point than the dot on the servo axis



Warning 3: adjusts the three dots of the motors with three dots of structural parts.



Warning 4: use thread locker to prevent vibrations from untying the screws. However, dip the extremity of the screw on the thread locker is enough (a drop for each screw hole is too much), otherwise disassembling your robot can be very hard!

The assembly can be divided into seven major steps:

- Trunk ([Text instruction](#)) Videos demonstration [1/5](#), [2/5](#), [3/5](#), [4/5](#), [5/5](#).
- Chest ([Text instruction](#)) ([Video demonstration](#))
- Right Arm ([Text instruction](#)) Videos demonstration [1/4](#), [2/4](#), [3/4](#), [4/4](#).
- Left Arm, which is the exact symmetry of the Right Arm([Text instruction](#)) Videos demonstration [1/4](#), [2/4](#), [3/4](#), [4/4](#).
- Legs (only for Humanoid version) ([Text instruction](#))([Video demonstration](#))
- Legs + Trunk (only for Humanoid version) ([Text instruction](#)) ([Video demonstration](#))
- Head ([Text instruction](#))

As previously said, this section was just intended to give you an idea about what building a Torso or a Humanoid implies.

2.1 Start and connect the robot

In this section, we will describe how to start your robot and give an overview of the possibilities to connect to it. We will focus here on real robots but we will also point to the advanced sections to follow if you are using a simulated robot.

2.0.1 Setup the software

Poppy creatures come with an embedded board which job is to control motors and access the sensors. This computer can be accessed through a web interface which makes it easy to control the robot from your own computer or a tablet without having to locally install anything specific.

There are two ways to setup the board for your Poppy:

- **the easy way:** use a pre-made ISO image of the Poppy operating system and write it to an SD-card
- **the hard way for advanced users:** install everything from scratch

Note: If you are using a simulated robot, you must install the software locally. Thus, you can not use a pre made SD-card. You will have to follow steps from the advanced way adapted to your configuration (OS, architecture...). The Chapter [TODO USE SIMU](#) will guide you through all the detailed steps.

2.0.1.1 Easy way: use the Poppy SD-card

The easiest and quickest way - by far - is to use an already made system image for a SD-card. ISO images come with everything pre-installed for your Poppy robot. It is also a good way to ensure that you are using exactly the same software as we are. Thus, you will avoid most problems.

The images can be found in the [github of the project](#):

- [for the ErgoJR](#)
- [for the Torso](#)
- [for the Humanoid](#)

They can be written to a SD-card (at least 8 Go) by using classical utility tools. Details can be found on the procedure of One the SD-card is ready, just insert it into the board and when you plug your robot it should automatically start and you should then be able to connect to the web interface.

2.0.1.2 Advanced way: DIY, install everything from scratch

The other way is basically to install everything needed from scratch. This basically means follow the same procedure as we are, to generate the image for SD-card. This can be useful if:

- You are **working with a simulated robot** and thus have to manually install all the required software on your computer, this procedure could be a good place to see how this can be done on a Raspberry-Pi and adapted to another computer,
- you want to customize the environment,
- or simply if you like to understand how it works.

Warning: We try to keep this installation procedure as generic as possible. Yet, some details may vary depending on your operating system or your computer. Moreover, the installation from scratch required some good knowledge of how to install and setup a python environment.



The entire procedure is detailed in the [Installation for advanced users](#). Depending on what you want to do all steps are not necessary required. In particular, you may just want to install the python libraries for Poppy, or entirely setup a Linux environment to match the Poppy ones. The entire installation process used to make the SD-card image can be sum up with those steps:

- Customize a [Raspbian](#) install for Poppy (setup a user/hostname...).
- Setup the Poppy utility tools (for install/update)
- Install the [Python Anaconda distribution](#).
- Install the [python libraries](#)) for Poppy.
- Setup the sensors for your creature.
- Install the web server for the control and monitoring interface.

2.0.2 Setup the network

Once your Poppy is built and its software is ready, the next step is to connect it to a network so you can start access it from your computer or smartphone/tablet and then control and program it.

Indeed, except if you plug a keyboard and a screen to the embedded board you will not be able to directly use it. Thus, you have to connect to it to your network so you can remotely access it. This section will describe the main steps to follow to do that.

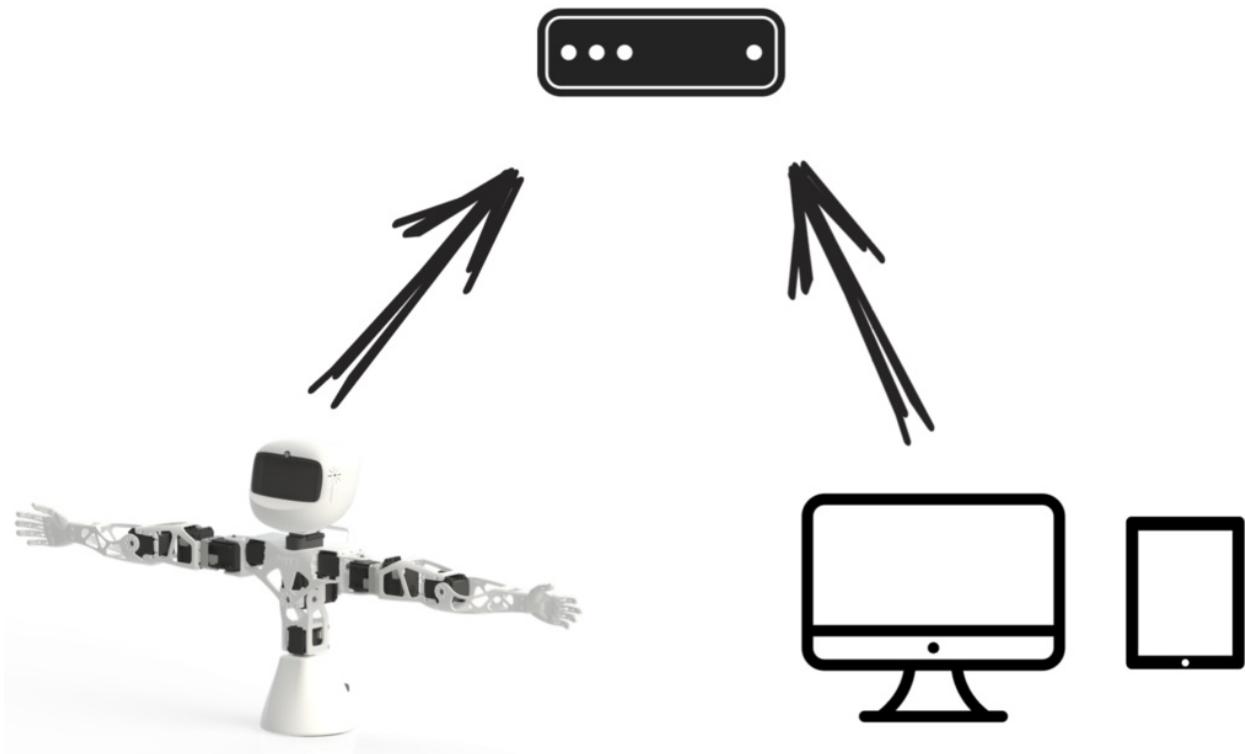
Poppy robots can use either Ethernet (using cable) or wifi network. Yet, you first need to use an Ethernet network so you can first configure the robot so it can then automatically use your specified wifi.

Poppy robots can use either Ethernet or wifi network. But first, plug the robot and your computer on the same Ethernet network.

Then, you need to choose among the two different possibilities:

- Use the [Zeroconf protocol](#) to connect to the robot using its hostname: "*poppy.local*". This should work directly under Mac OS and GNU/Linux but required to install [Bonjour Print Services](#) on Windows. This is the simplest way and should be preferred except if you do not have the administrator right on your Windows computer for instance. Note that *Bonjour Print Services for Windows* may be already installed if you have iTunes or QuickTime on your computer.
- Know the IP address of the robot assigned by the router of your network.

Note: If you are not familiar with network configuration or have no idea what the previous paragraph clumsily tried to say, you should see with the IT network engineer, how you can do that.



2.0.3 Use the web interface

Once you managed to have either Zeroconf working or the IP address of your robot, you can directly access the control and monitoring interface. This web interface can be used to:

- setup your wifi and change the name of the robot - the one you will use to connect,
- to launch demo primitives,
- and monitor the state of your robot (if motors are too hot for instance).

To access this interface, you simply have to go to the URL using your favourite web browser:

- <http://poppy.local> (if you changed the name of your robot, simply replace *poppy* by its new name)
- or using directly its IP address, something similar as <http://192.168.0.42>

You should see something like:

Control Poppy

- [Configure wifi and change hostname](#)
- [Open Ipython notebook](#): to control Poppy with Python code
- [Start Snap!](#), a block programmation webapp. It works faster with chromium/chrome browser
- [Start Poppy-monitor](#): a web app to easy start primitives and looking state of motors

Debug

- [Kill all acces to motors](#)
- [Display logs](#)

And then if you click on the **Start Poppy-monitor** link:

Monitoring Poppy

Activer tous les moteurs Désactiver tous les moteurs Arrêter toutes les primitives

bpm : 120 Update
amplitude : 20 Update

head_y: 0°C @ 2.7°
head_z: 0°C @ -71.0°
r_shoulder_y: 0°C @ 65.0°
r_shoulder_x: 0°C @ 15.2°
l_shoulder_y: 0°C @ 13.0°
l_shoulder_x: 0°C @ -16.2°
r_elbow_y: 0°C @ -52.1°
l_elbow_y: 0°C @ 0.7°
r_elbow_x: 0°C @ 16.0°
l_elbow_x: 0°C @ -76.3°
r_bust_y: 0°C @ -24.7°
l_bust_y: 0°C @ 14.8°
r_bust_x: 0°C @ 14.8°
l_bust_x: 0°C @ -16.2°
r_abs_y: 0°C @ 29.1°
l_abs_y: 0°C @ -29.8°
r_abs_x: 0°C @ -40.1°
l_abs_x: 0°C @ 50.7°
r_hip_y: 0°C @ 37.9°
l_hip_y: 0°C @ -21.4°
r_hip_x: 0°C @ 71.7°
l_hip_x: 0°C @ 12.6°
r_knee_y: 0°C @ 30.1°
l_knee_y: 0°C @ 56.2°
r_knee_x: 0°C @ -43.5°
l_knee_x: 0°C @ -5.9°

Get motor/list.json
Get sensor/list.json
Get sensor/alias.json
Get motor/motor_name/register/list.json
Get sensor/motor_name/register/list.json
Get motor/motor_name/register/register_name
Get sensor/motor_name/register/register_name
Post motor/motor_name/register/register_name/value.json
Post sensor/motor_name/register/register_name/value.json
Get primitive/list.json
Get primitive/running/list.json
Get primitive/primin/start.json
Get primitive/primin/stop.json
Get primitive/primin/pause.json
Get primitive/primin/resume.json
Get primitive/primin/property/list.json
Get primitive/primin/property/prop
Post primitive/primin/property/prop/value.json
Get primitive/primin/method/list.json
Post primitive/primin/method/method/args.json

3.1 Program the robot

In this section we will present the different options to program your robot ranging from the high level behaviors triggered using [Snap!](#), the visual programming language, to the low-level API written in [Python](#).

We will also say a few words on how you can interface Poppy robots with other devices or any programming language using [Poppy's REST API](#).

As for the rest of the project, all our libraries are open source and available on [GitHub](#).



3.0.1 Using Snap!

3.0.2 Using Python



Poppy libraries have been written in Python, to allow for fast development and extensibility and to benefit from all existing scientific libraries. Python is also a well-known language and widely used in education or artistic fields. By programming Poppy in Python, you will have access from the very low-level API to the higher levels.

The API has been designed to allow for very fast prototyping. In particular, creating a robot and starting to move motors should not take more than a few lines:

```
from poppy.creatures import PoppyErgoJr

jr = PoppyErgoJr()
jr.m3.goal_position = 30
```

We are also big fan of the [Jupyter Project](#) and notebooks. Notebooks are documents which contain both Python code and rich text elements like equations, pictures, videos. They can be edited from the Jupyter Web interface which allow users to program Poppy robots directly from a website hosted on the robot computer. We really think that this is an extremely powerful tool permitting the creation and sharing of live code, results visualizations and explanatory text combined in a single document.



Most of the tutorials, experiments or pedagogical activities that we and the community develop are available as [notebooks](#).

 A screenshot of a Jupyter Notebook window. The title bar says "jupyter Untitled Last Checkpoint: 9 minutes ago (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Help. The toolbar shows icons for file operations and a "Python 2" kernel. The main content area has a heading "Discover the Poppy Ergo Jr" and a sub-section "Create the Robot in Python". It contains text about importing the class and creating an instance, followed by two code snippets in a code cell:


```
In [ ]: from poppy.creatures import PoppyErgoJr
```

 Then, you can directly "instantiate" it:


```
In [ ]: ergo_jr = PoppyErgoJr()
```

 Below the code cell, there is an image of a Poppy Ergo Jr robot arm holding a white lamp.

An update gallery of notebooks can be found [here](#).

3.0.3 Through the REST API

On top of the Snap! and Python options, it was really important for us to provide another way of accessing and controlling your robot accessible from any device or language. Thus, all our Poppy robots are providing a REST API, meaning that the most important features of the robot can be access through HTTP GET/POST requests.

From a more practical point of view, this allows to:

- **Write bridges to control Poppy robot in any language** (awesome contributors have already written [Matlab](#) and [Ruby](#) wrappers).
- Easily **design web apps** connect to your robot (such as the [monitor interface](#)).
- Make your **robot interact with other connected device** such as a smartphone, intelligent sensors, or even your twitter account...

4.1 Visualize the robot in a simulator

4.0.1 Simulated Poppy Creatures

Simulated versions of all Poppy robots (Humanoid, Torso, and Ergo Jr) are also available.

At the moment, only Poppy Ergo Jr can be used in the web visualizer. If you want to simulate other creatures, you should use V-REP.

Note: If you want to use Poppy robots using a simulator you will have to install some of the poppy libraries locally on your computer.

Indeed, while those libraries are already installed within the physical robots, they are not integrated in the simulators. More detailed of what you will have to do is given in the [section below](#).

Moreover, you will not have access to the robot web interface, but you will have to manually launch the different services to start programming your robot (the Jupyter server for Python notebooks, or use the Snap! server).

This allows the development and test of programs without having a real robot. This is particularly useful:

- To discover and try the robot possibilities without having to spend money.
- In a context where multiple users share a robot; for instance in a classroom where each group can work using the simulator and validate their program on the real robot.
- To design and run complex and time consuming experiments.

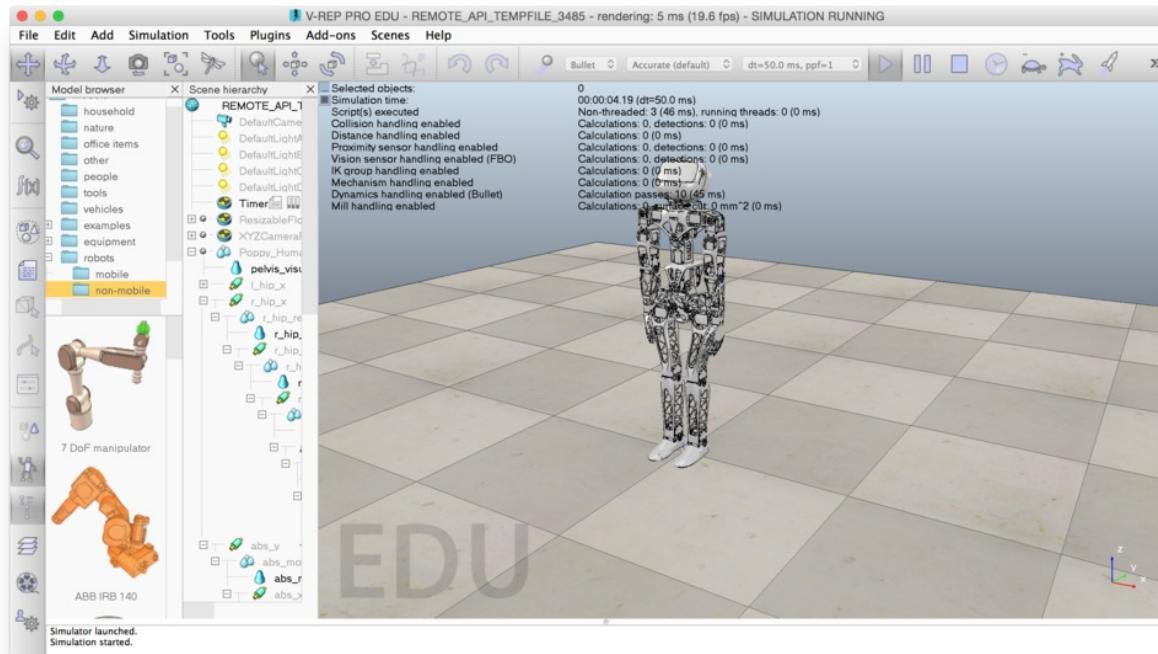
We try to make the switch from a simulated Poppy robot to the real one as transparent and as simple as possible. Most of the documentation is actually valid for both simulated and real robots. Yet, a dedicated chapter [Use a simulated robot](#) will describe in details everything you need to know and do to use a simulated Poppy robot.

Once your computer is setup, you will be able to write a program for the simulator and directly use it on a real robot (or vice-versa).

4.0.2 Using V-REP

V-REP is a well known and powerful robot simulator. It is widely used for research and educational purposes. Moreover, it is available for free under a educational license.

It is important to note that as V-REP is simulating the whole physics and rendering of the robot, it may be slow if you do not have a recent computer (especially the GPU card).



All main Poppy robots are available in V-REP:

- Poppy Humanoid
- Poppy Torso
- Poppy Ergo Jr

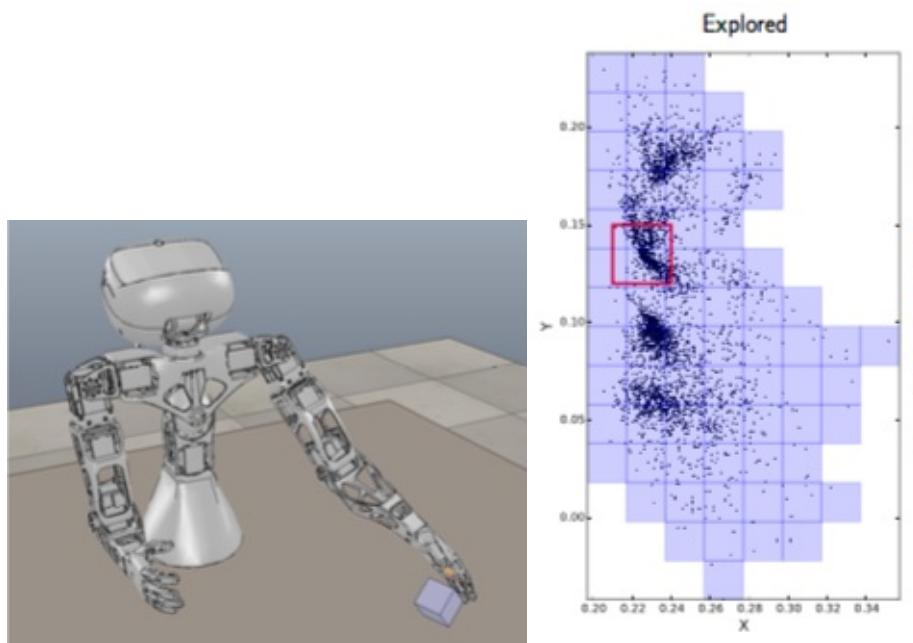
To start using V-REP for controlling Poppy robots, you will need:

- to install V-REP (work under Mac OS, Windows and GNU/Linux)
- to have a working Python (version ≥ 2.6 or ≥ 3.0), we strongly recommend to use the [Anaconda Python distribution](#).
- to install the Poppy libraries: pybot and the library corresponding to your creature (e.g. poppy-ergo-jr)

Note: Details about all those steps can be found in section [Install everything locally for using a simulator](#).

V-REP can be used to learn how to control motors, get information from sensors but also to interact with the simulated environment. It can be used using Python, Snap! or through the REST API. Here, are some examples of what the community has already been doing using it:

- In a pedagogical activity to discover the different motor of your robot and how they can be controlled.
- In a scientific experiment, where a Poppy Torso is learning how to push a cube on a table in front of it



Note: Even if we try, to reproduce the robot behaviour and functioning, some differences remain. In particular, if you make a robot walk in simulation that does not necessarily mean that it will walk in the real world.

5.1 Overview of projects developed by the community

7Setup your Poppy Board

Note: this chapter is only for people who want to control a tangible robot. If you intend to control a simulated robot on your computer, go to the [installation for advanced users section](#)

Poppy creatures are controlled by a small embedded computer: a Raspberry Pi or a Odroid for the older versions.

The Operating System of this computer is hosted on a SD card.

You may be in two kind of cases:

- You already have a SD-card with the Poppy Operating System (provided by one of the Poppy distributors for example). You just have to put the SD card on the board.
- You have an empty SD-card, so you have to install the operating System on the SD-card (explained below) and put the SD-card on the board.

Note: The Poppy creatures Operating System use a GNU/Linux distribution, but you won't have to any knowledges on Linux to install the image on the Raspberry Pi. You will only need a computer with a SD card reader to write the image on the SD card.

7.1Download the image

You have to choose the image to download depending on your Poppy creature and the targeted board:

- [Poppy ErgoJr](#)
- [Poppy Torso](#)
- [Poppy Humanoid](#)

The .zip file you have downloaded need to be unzipped to get the image file for writing to your SD card.

7.2Write an image to the SD card

With the image file corresponding to your Poppy creature, you need to use an image writing tool to install it on your SD card.

See our guide for your system:

- [Windows](#)
- [Mac OS](#)
- [GNU/Linux](#)

7.0.1 Installing Poppy Operating System Images from a Windows computer

- Insert the SD card into your SD card reader and check which drive letter was assigned. You can easily see the drive letter (for example `G:`) by looking in the left column of Windows Explorer. You can use the SD Card slot (if you have one) or a cheap SD adaptor in a USB port.
- Download the Win32DiskImager utility from the [Sourceforge Project page](#) (it is also a zip file); you can run this from a USB drive.
- Extract the executable from the zip file and run the `Win32DiskImager` utility; you may need to run the utility as administrator. Right-click on the file, and select **Run as administrator**.
- Select the image file you extracted above.
- Select the drive letter of the SD card in the device box. Be careful to select the correct drive; if you get the wrong one you can destroy your data on the computer's hard disk! If you are using an SD card slot in your computer and can't see the drive in the Win32DiskImager window, try using a cheap SD adaptor in a USB port.
- Click `Write` and wait for the write to complete.
- Exit the image and eject the SD card.

8.0.1Installing Poppy Operating System Images from a Mac OS computer

On Mac OS you have the choice of the command line `dd` tool or using the graphical tool ImageWriter to write the image to your SD card.

8.0.1(Mostly) graphical interface

- Connect the SD card reader with the SD card inside. Note that it must be formatted in FAT32.
- From the Apple menu, choose About This Mac, then click on More info...; if you are using Mac OS X 10.8.x Mountain Lion or newer then click on System Report.
- Click on USB (or Card Reader if using a built-in SD card reader) then search for your SD card in the upper right section of the window. Click on it, then search for the BSD name in the lower right section; it will look something like 'diskn' where n is a number (for example, disk4). Make sure you take a note of this number.
- Unmount the partition so that you will be allowed to overwrite the disk; to do this, open Disk Utility and unmount it (do not eject it, or you will have to reconnect it). Note that On Mac OS X 10.8.x Mountain Lion, "Verify Disk" (before unmounting) will display the BSD name as "/dev/disk1s1" or similar, allowing you to skip the previous two steps.
- From the terminal run `sudo dd bs=1m if=path_of_your_image.iso of=/dev/rdiskn`. Remember to replace `n` with the number that you noted before. If this command fails, try using `disk` instead of `rdisk`: `sudo dd bs=1m if=path_of_your_image.iso of=/dev/diskn`.

8.0.0.1Command line

- If you are comfortable with the command line, you can write the image to a SD card without any additional software. Open a terminal, then run:

```
diskutil list
```

- Identify the disk (not partition) of your SD card e.g. `disk4` (not `disk4s1`).
- Unmount your SD card by using the disk identifier to prepare copying data to it:

```
diskutil unmountDisk /dev/disk<disk#> from diskutil>
```

e.g. `diskutil unmountDisk /dev/disk4`

- Copy the data to your SD card:

```
sudo dd bs=1m if=image.img of=/dev/rdisk<disk#> from diskutil>
```

e.g. `sudo dd bs=1m if=poppy-ergojr.iso of=/dev/rdisk4`

- This may result in an `dd: invalid number '1m'` error if you have GNU coreutils installed. In that case you need to use `1M`:

```
sudo dd bs=1M if=image.img of=/dev/rdisk<disk#> from diskutil>
```

This will take a few minutes, depending on the image file size. You can check the progress by sending a `SIGINFO` signal pressing `Ctrl+T`.

- If this command still fails, try using `disk` instead of `rdisk`:

```
sudo dd bs=1M if=poppy-ergojr.iso of=/dev/disk4
```

or

```
sudo dd bs=1M if=poppy-ergojr.iso of=/dev/disk4
```

8.0.0.2Alternative method

These commands and actions need to be performed from an account that has administrator privileges.

- From the terminal run `df -h`.
- Connect the SD card reader with the SD card inside.
- Run `df -h` again and look for the new device that wasn't listed last time. Record the device name of the filesystem's partition, for example `/dev/disk3s1`.
- Unmount the partition so that you will be allowed to overwrite the disk:

```
sudo diskutil unmount /dev/disk3s1
```

(or open Disk Utility and unmount the partition of the SD card (do not eject it, or you will have to reconnect it)

- Using the device name of the partition, work out the raw device name for the entire disk by omitting the final "s1" and replacing "disk" with "rdisk". This is very important as you will lose all data on the hard drive if you provide the wrong device name. Make sure the device name is the name of the whole SD card as described above, not just a partition of it (for example, `rdisk3`, not `rdisk3s1`). Similarly, you might have another SD drive name/number like `rdisk2` or `rdisk4`; you can check again by using the `df -h` command both before and after you insert your SD card reader into your Mac. For example, `/dev/disk3s1` becomes `/dev/rdisk3`.
- In the terminal, write the image to the card with this command, using the raw disk device name from above. Read the above step carefully to be sure you use the correct rdisk number here:

```
sudo dd bs=1m if=poppy-ergojr.iso of=/dev/rdisk3
```

If the above command reports an error (`dd: bs: illegal numeric value`), please change `bs=1m` to `bs=1M`.

If the above command reports an error `dd: /dev/rdisk3: Permission denied` then that is because the partition table of the SD card is being protected against being overwritten by MacOS. Erase the SD card's partition table using this command:

```
sudo diskutil partitionDisk /dev/disk3 1 MBR "Free Space" "%noformat%" 100%
```

That command will also set the permissions on the device to allow writing. Now try the `dd` command again.

Note that `dd` will not feedback any information until there is an error or it is finished; information will be shown and the disk will re-mount when complete. However if you wish to view the progress you can use 'ctrl-T'; this generates SIGINFO, the status argument of your tty, and will display information on the process.

- After the `dd` command finishes, eject the card:

```
sudo diskutil eject /dev/rdisk3
```

(or: open Disk Utility and eject the SD card)

9.0.1Installing Poppy Operating System Images from a GNU/Linux computer

9.0.0.1Graphical interface

9.0.0.1.1With Any distribution (Ubuntu, Fedora, Arch, Debian, ...)

- Download [UNetbootin binaries](#).
- To run these binaries go to Right Click->Properties->Permissions and check "Execute"), or run `chmod +x ./unetbootin-linux` in a Terminal then start the application by running `./unetbootin-linux`.
- Select the Poppy image in the DiskImage section.
- Select the drive matching your SD card.
- Click "ok" and be patient.

9.0.0.1.2If you are using Linux Mint

A program called "Usb key creator" is already installed and allows you to do the same operations than above.

9.0.0.2Command line

Please note that the use of the `dd` tool can overwrite any partition of your machine. If you specify the wrong device in the instructions below, you could delete your primary Linux partition. Please be careful.

- Run `df -h` to see what devices are currently mounted.
- If your computer has a slot for SD cards, insert the card. If not, insert the card into an SD card reader, then connect the reader to your computer.
- Run `df -h` again. The new device that has appeared is your SD card. The left column gives the device name of your SD card; it will be listed as something like `/dev/mmcblk0p1` or `/dev/sdd1`. The last part (`p1` or `1` respectively) is the partition number but you want to write to the whole SD card, not just one partition. Therefore you need to remove that part from the name (getting, for example, `/dev/mmcblk0` or `/dev/sdd`) as the device for the whole SD card. Note that the SD card can show up more than once in the output of `df`; it will do this if you have previously written a Raspberry Pi image to this SD card, because the Raspberry Pi SD images have more than one partition.
- Now that you've noted what the device name is, you need to unmount it so that files can't be read or written to the SD card while you are copying over the SD image.
- Run `umount /dev/sdd1`, replacing `sdd1` with the device name of your SD card (including the partition number).
- If your SD card shows up more than once in the output of `df` due to having multiple partitions on the SD card, you should unmount all of these partitions.
- In the terminal, write the image to the card with the command below, making sure you replace the input file `if=` argument with the path to your `.img` file, and the `/dev/sdd` in the output file `of=` argument with the right device name.
This is very important, as you will lose all data on the hard drive if you provide the wrong device name. Make sure the device name is the name of the whole SD card as described above, not just a partition of it; for example `sdd`, not `sdds1` or `sddp1`; or `mmcblk0`, not `mmcblk0p1`.

```
dd bs=4M if=poppy-ergojr.iso of=/dev/sdd
```

- Please note that block size set to `4M` will work most of the time; if not, please try `1M`, although this will take considerably longer.
- Also note that if you are not logged in as root you will need to prefix this with `sudo`.
- The `dd` command does not give any information of its progress and so may appear to have frozen; it could take more than five minutes to finish writing to the card. If your card reader has an LED it may blink during the write process. To see the progress of the copy operation you can run `pskill -USR1 -n -x dd` in another terminal, prefixed with `sudo` if you are not logged in as root.

The progress will be displayed in the original window and not the window with the `pkill` command; it may not display immediately, due to buffering.

- Instead of `dd` you can use `dcf1dd`; it will give a progress report about how much has been written.
- Run `sync`; this will ensure the write cache is flushed and that it is safe to unmount your SD card.
- Remove the SD card from the card reader.

12 Assembly guide for the Ergo Jr



The Poppy Ergo Jr robot is a small and low cost 6-degree-of-freedom robot arm. It consists of very simple shapes which can be easily 3D printed. They are assembled via OLLO rivets which can be removed and added very quickly with the OLLO tool.

Its end effector can be easily changed. You can choose among several tools:

- a lampshade,
- a gripper,
- or a pen holder.

Thanks to the rivets, they can be very quickly and easily swapped. This allows the adaptation of the tooltip to the different applications you plan for your robot.

The engines have the same functionality as other Poppy creatures but are slightly less powerful and less precise. The advantage being that they are also less expensive.

The electronic card is visible next to the robot, which is very advantageous to understand, manipulate, and plug extra sensors. No soldering is needed, you just need to add the shield for XL-320 motors on top of the Raspberry Pi pins.

This chapter will guide you through all steps required to entirely assemble a Poppy Ergo Jr. It will cover:

- [motors configuration](#)
- [electronic assembly](#)
- [hardware construction](#)

The entire assembly should take about one or two hours for the first time you build one. With more practice, half an hour should be more than enough.

At the end of the process, you should have a working Poppy Ergo Jr, ready to move!

We recommend you to follow carefully the instructions. Even if the Ergo Jr can be easily disassembled, it is always disappointing to need to start from scratch again because you forgot to configure the motors, a wire is missing, or a motor is reversed.

13 Bill of materials

Here you will find the complete list of material (BOM) needed to build a Poppy Ergo Jr.

13.1 Poppy's material

- 1x [Pixel board](#) (*electronic board to control XL320 motors*)
- 1x [disk_support.stl](#) (using laser cutting) the 2D plan can be found [here](#). You can also 3D print the base but it will take a lot of time
- the 3D printed parts [STL here](#)
 - 1x [base.stl](#)
 - 3x [horn2horn.stl](#)
 - 3x [side2side.stl](#)
 - 1x [long_U.stl](#)
 - 1x [short_U.stl](#)
 - 1x [support_camera.stl](#)
 - the different tools
 - 1x [lamp.stl](#)
 - 1x [gripper-fixation.stl](#)
 - 1x [gripper-fixed_part.stl](#)
 - 1x [gripper-rotative_part.stl](#)
 - 1x [pen-holder.stl](#)
 - 1x [pen-screw.stl](#)

13.2 Robotis parts

- 6x Robotis dynamixel motors XL-320
- 1x set of OLLO rivets (about 70 colored and 4 grey)
- 1x OLLO TOOL
- 4x M2.5x6mm screw (for fixing the Raspberry Pi on the base)

13.3 Various electronics

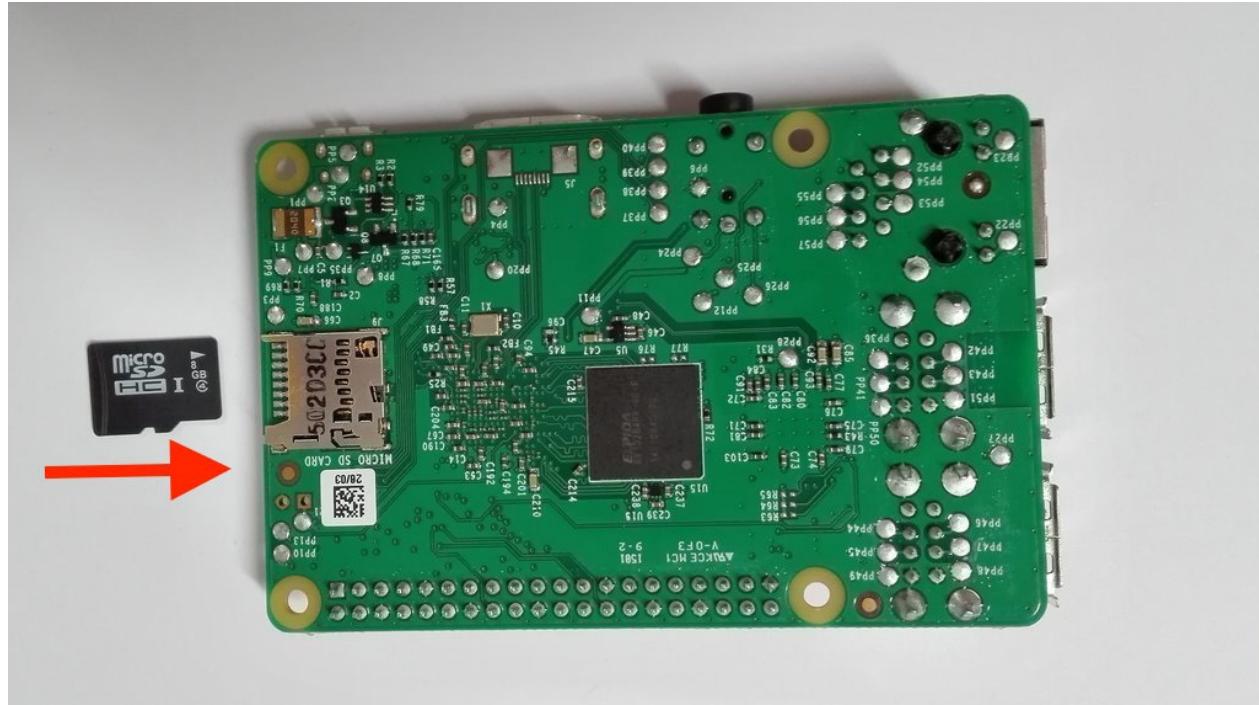
- 1x Raspberry Pi 2 8Go
- 1x Raspberry Pi camera
- 1x AC power 7.5V 2A with a 2.1 x 5.5 x 9.5 jack connector ([this one](#) for instance).

12.1 Electronic assembly

12.0.0.1 Insert the micro-SD card in the Raspberry Pi

Make sure you use a pre-configured micro SD-card. If it not the case, you have to "burn" your micro-SD card with the ergo-jr ISO image, this is described in the [startup section](#).

Insert the micro-SD card inside the Raspberry Pi: push the micro-SD in the connector slot until you hear a "click" sound.



12.0.0.1 Assemble the pixel board

[Pixel bord](#) allow to power up the Raspberry Pi from 7.5V DC power or battery and communicate with XL-320 motors.

Plug the pixel at the end of Raspberry Pi headers.



Once the pixl is plugged (**and not before**), you can plug the DC power and the motors wire.



You need to switch off the power supply of the Pixl board before to put in or to take off the Pixl board of the Raspberry pi. Otherwise, you risk to burn the power converter of the Pixl board.

You can now [configure your motors](#).

13.1 Motor configuration

The Ergo Jr is made of 6 XL-320 motors from [Robotis](#). Each of this servomotor embeds an electronic board allowing it to receive different kind of orders (about position, speed, torque...) and to communicate with other servos. Therefore, you can chain up several of this servomotors and command them all from one end of the chain: each servomotor will pass the orders to the next one.



Yet, in order for the motors to be connected and identified on the same bus (same line), they must have a unique ID. Out of the factory they all set to the same ID: 1. In this section, we will give you details on how you can set a new and unique ID to each of your motors.

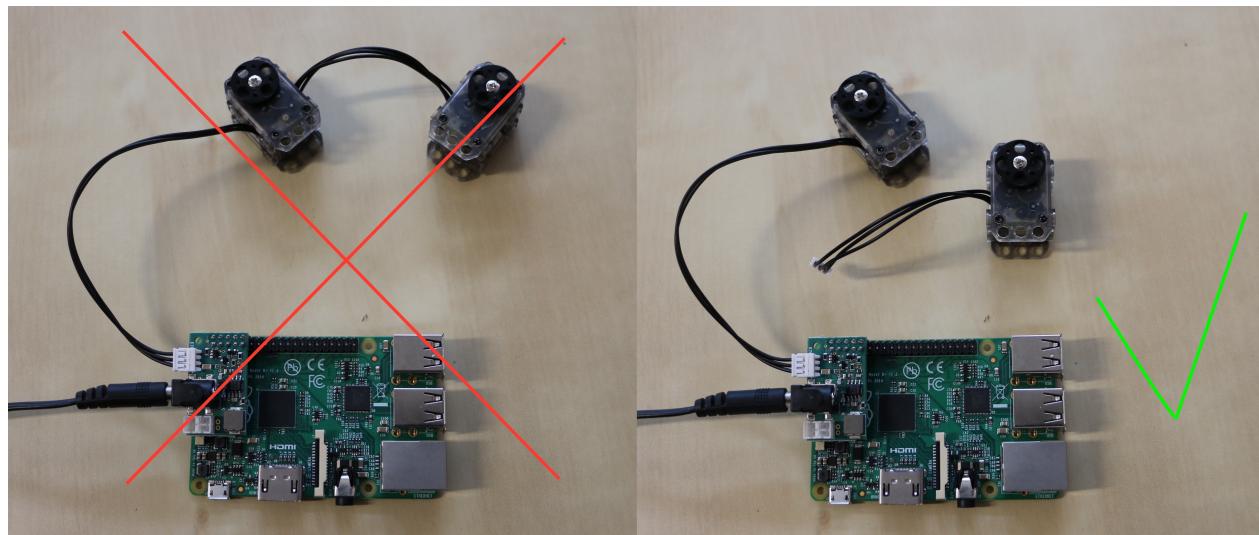
We recommend to configure motors in parallel of the hardware assembly. Meaning, that before assembling a new motor, you first configure it, then assemble to the rest of your robot. This will prevent you to swap motors. In the step-by-step assembly procedure, we will point out each time you need to configure a new motor. Furthermore, you will also be able to directly configure the motor from the assembly notebook interface.

13.0.1 Configuring motors one at a time

As explained above, all motors have the same ID by default. **Only one motor at a time should be connected to the data bus when you configure them.** Otherwise, it will not work as all motors connected will think that the order sent on the line is intended for them, they will all try to answer resulting in a big mess.

Your electronic setup when configuring a motor should look like this:

- the Raspberry Pi
- the shield on top and the AC plugged
- a wire from the shield to the motor you want to configure



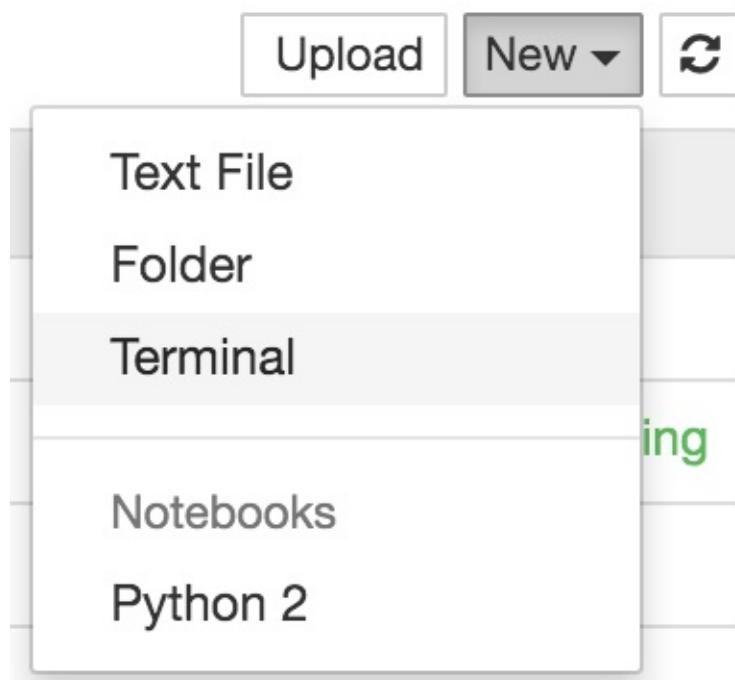
13.0.1.1 Web interface (user friendly)

The web utility for configuring motors is still in the TODO stack.

13.0.1.2 Command-line utility

Robots come with a command line utility `poppy-configure` ; to use it you need to open a terminal on your Raspberry Pi.

You can open a terminal on your robot directly from your Ipython/Jupyter notebook. For that, open the home page of the robot in your web browser (eg. open <http://poppy.local> if the hostname is poppy). Click on the "Open Ipython/Jupyter Notebook window" link and



select "New Terminal".

Once the terminal is open, copy and press enter to execute the command below.

```
poppy-configure ergo-jr m1
```

You have now configured the m1 motor of your robot. Once configured - either from the web interface or the command line - and that you see the message indicating that everything went well, you can unplug the motor. Its configuration will be stored in the motor internal memory.

Poppy Ergo Jr motors are named m1, m2, m3, m4, m5, m6. To configure the others motors, you have to change m1 by the name of the motor you want to configure in the command above.

14.1 Mechanical Assembly

14.0.1 General advices and warnings

- You can assemble all the rivets before the construction. You have to put the edges of the first part in the second part holes. You will thus be able to remove them easily if needed.



Part 1



Part 2



Part 3

- There are two kinds of rivets. The grey ones and the others. Grey rivets are longer to be able to be inserted in the motor axis, at the opposite side of the horn.



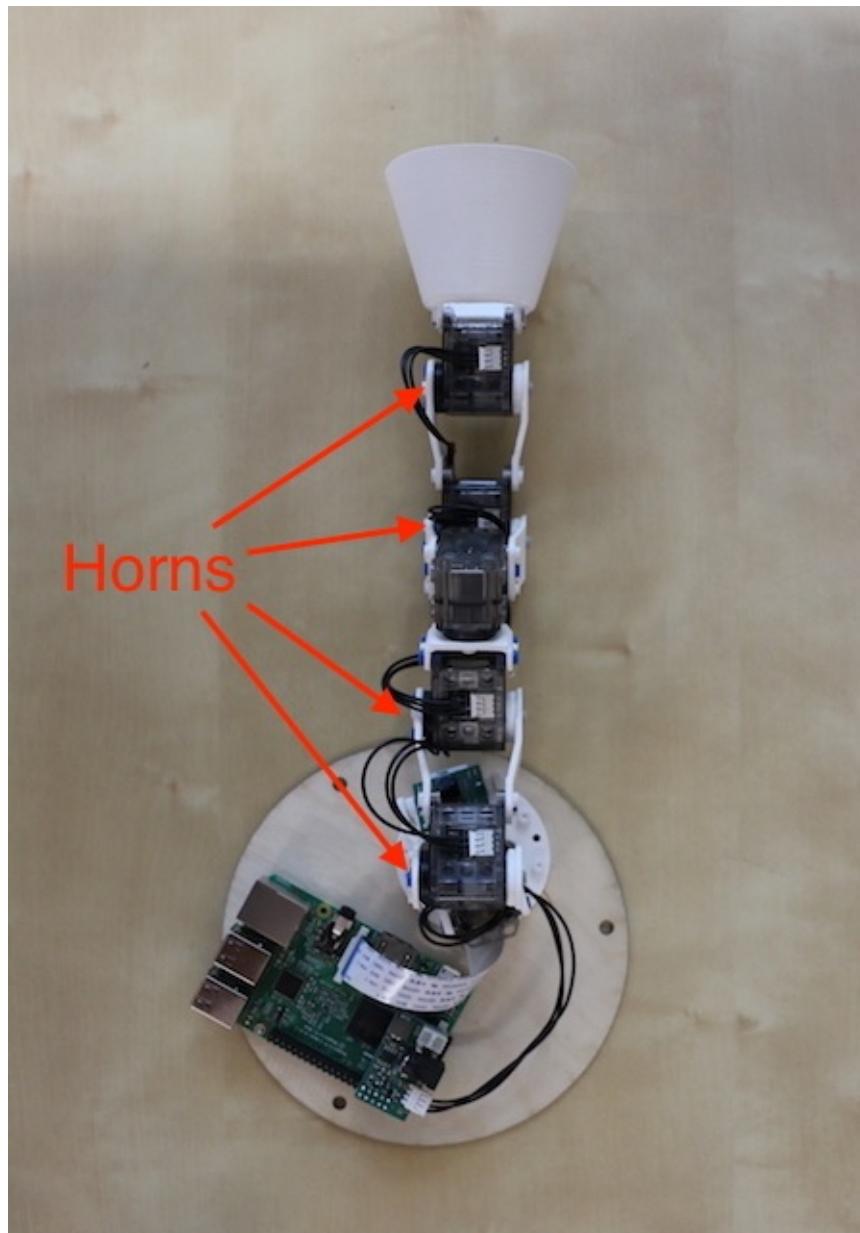
- Use the OLLO Tool for putting and removing rivets easily.



- Do not forget to put wires between motors while building the robot! Each motor, except the last, must have two wires; one connected to the previous motor and the other to the next (no favourite side).
- **Always align the horn with the motor before assembling them!** Otherwise your Poppy Ergo Jr will look all weird.



- Every motor horns (black revolving circle) are **facing the left side of the robot**. It is a convention but it will define the orientation of your motors.



14.1 Step-By-Step guide

14.0.1 Motor configuration (for all steps)

You can configure your motors before, during or after the the mechanical assembly but it is highly advised to configure each motor one by one in the construction order :

- configure motor m1
- assemble the base and motor m1
- configure motor m2
- ...

To configure motors, you have to connect them separately one by one to the Raspberry Pi. If you try to configure a new motor wired to a previously configured motor, this will not work.

Please consult the [motor configuration section](#) for more informations.

14.0.2 Step 1

First, [configure one XL-320 motor as "m1"](#).

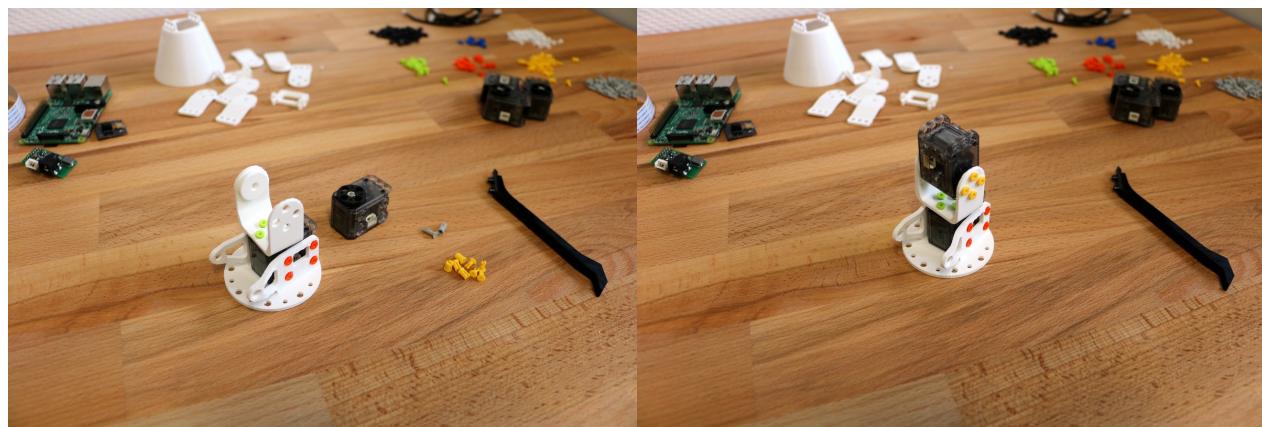
Mount the motor on the 3D printed base.



14.0.3 Step 2

Configure the second motor, its name is "m2".

Mount the *long_U* part. Be careful of the orientation of the U, the horn must be oriented in the left. Mount the motor "m2" on top of the construction.



14.0.4 Step 3

Configure a third motor: "m3".

Mount *horn2horn* and *horn2side* parts on motor "m2", and mount "m3" on top of the construction.



14.0.5 Step 4

Configure the fourth motor: "m4".

Mount the *short_U* on it.



Mount motor "m4" and the assembled *short_U* on top of the previous assembly. The nose of the motor should be on the other side of the base.



14.0.6 Step 5

Configure the fifth motor: "m5".

Mount *horn2horn* and *horn2side* parts on motor "m4", and mount "m5" on top of the construction.



14.0.7 Step 6 - the tool of your choice

Configure the sixth motor: "m6".

To finish your Ergo Jr, you need to add a tool at its end. So first choose the tool you want depending on what you want to do.

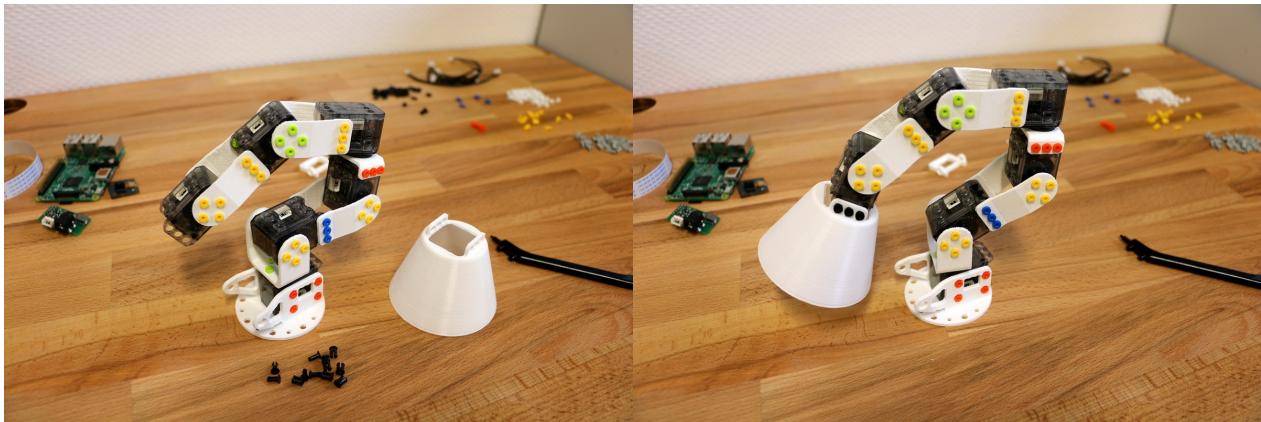
Tools they can be easily and quickly changed, so you can adapt it to the different activities.

14.0.7.1 Lampshade and pen holder

Mount *horn2horn* and *horn2side* parts on motor "m5", and mount "m6" on top of the construction.



You can mount the pen holder or the lampshade on the motor "m6".



14.0.7.2 Gripper

Mount the *gripper-fixation* between motors "m5" and "m6".

Mount *gripper-fixed_part* and *gripper-rotative_part* on motor "m6".

14.0.8 Step 7 - fix you ergo-jr to the base

Mount you ergo-jr to the wood disk-support.

Mount the Raspberry Pi to the disk support, and use 4 x M2.5x6mm screw to fix it.

14.0.9 Step 8 - electronics

Mount the *support_camera* part on the base. Fix the Raspberry Pi camera on it and move the camera flex cable between motor "m1" and the base.



To fix the flex cable of the camera on the Raspberry Pi:

- open the camera connector by pulling on the tab
- make sure that connectors on the flex cable are facing away of the ethernet port

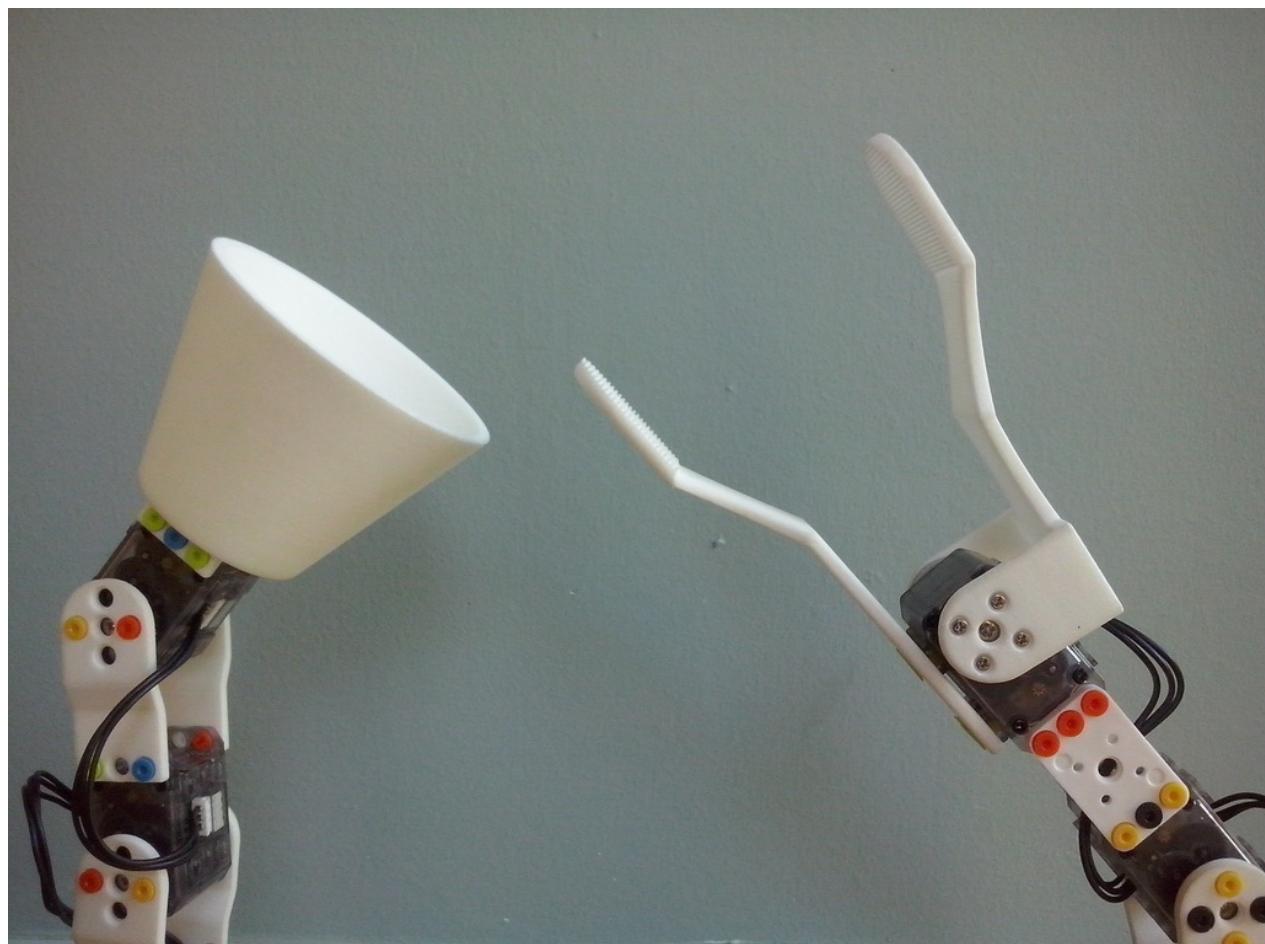
- push the flex on the port, and push the plastic tab down

Motors wires:

If it is not already done, you can plug every motors wires. Every motor has two connectors but there is no input or output: you just have to create a chain of motors. The first motor is linked to the pixl and the second motor; the last motor is linked only to the previous one, and every other motors are linked to the one above and ahead.

Connectors of the motor "m1" (in the base) are a bit hard to link, you can use the OLLO tool to help yourself.

14.0.10Done



Grab your [favorite drink](#) and relax.

16 Assembly guide for the Poppy Humanoid

The assembly guide for the humanoid has still not been merged on the new doc. You can find it on the [poppy-humanoid git repository](#).

19 Programming Poppy robots using Snap!

Snap! is a blocks-based graphical programming language that allows users to create interactive animations, games, and more, while learning about mathematical and computational ideas.

Snap! was inspired by Scratch (a project of the Lifelong Kindergarten Group at the MIT Media Lab), but also targets both novice and more advanced users by including and expanding Scratch's features.

Snap! is open-source and it is entirely written in javascript, you can use it from the [official website](#) but you can also use a [copy of the website](#) in your personal computer and open the `snap.html` file in your browser.

Note: Even if Snap! use JavaScript and HTML5 which are browser independent technologies, opening blocks for Poppy robots in Snap! is far faster in a web browser based on Webkit engine. We strongly recommend you to use [Chromium Browser](#)(which is very similar to Chrome without tracking tools), or Google Chrome.

19.1 Introduction to Snap! programming

This chapter will focus on things necessary to understand in Snap! for using Poppy creatures.

If you want a well designed online lesson on Snap! we strongly encourage you to look at the "[Beauty and Joy of Computing](#)" (BJC) course made by the University of Berkeley for New York high school students.

Some of the snapshots and concepts of BJC have been used for writing this chapter.

19.1.1 Connect your robot to Snap!

19.1.1.1 With a simulated robot with V-REP

19.1.1.2 With a tangible robot

Open the web interface of your robot in your web browser. Click on the "Start Snap! link"; it will start the python REST API to the robot and open the Snap! interface.

Click on the file icon -> Examples -> click on "pypot snap blocks" and the "open" button.

19.1.2 Interface and general ideas

19.1.3 Saving in Snap!

There are three ways of saving a project in Snap!

19.1.3.1 Save the project in your web browser



When you are not logged in Snap! Cloud, the default behaviour of Snap! is to save your project in **your browser**.

Technically this uses the Local Storage which is a memory space in your web browser where websites can store offline data. This is very convenient because you have not to register or to see Snap! project files, but keep in mind that **these projects are only visible in this specific web browser in this specific computer**.

19.1.3.2 Snap! Cloud

« There is no Cloud, it's just someone else's computer ».

Instead of saving your projects on your web browser, you can save them in Snap! servers in UC Berkeley, called "cloud". Moreover, this allows you to share your project with anyone, with a simple link.

19.1.3.2.1 Create an account on Snap! cloud

Click on the cloud button -> "signup...".



Fill the required fields in the modal window for signing up.



You will soon receive a validation email with a random password.

You can now log in with your username and password.



If you use your personal computer, remember to check the "stay signed in on this computer [...]" checkbox.



After logging in account, you are free to change your password: click on the cloud button -> "Change Password".

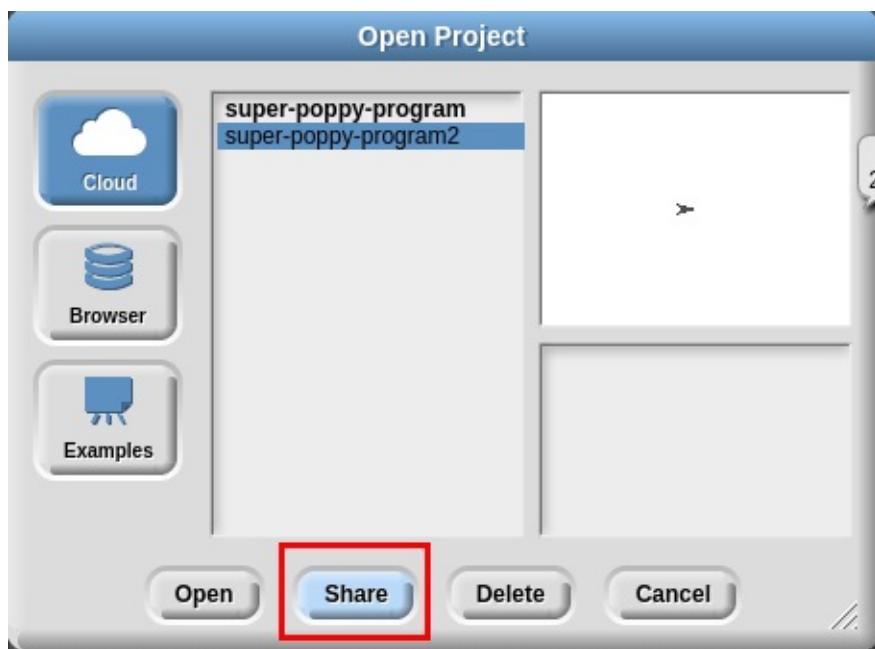


19.1.3.2 Share your Snap! project

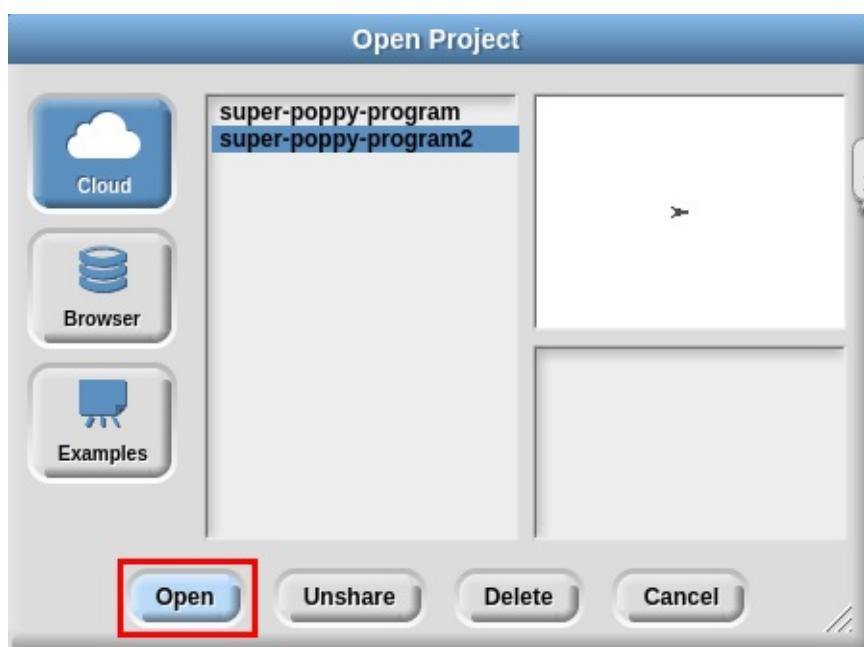
The big advantage of using Snap! Cloud is the ability to share a copy of your project with anyone. To share a Snap! project, you first need to be logged in Snap! Cloud and having your current project saved ("save" or "save as"). Go to the "open" menu:



In the cloud section, select the project you want to share and click on "Share" button.



Here is the trick step: to see the share link, you have to click on the "Open" button.

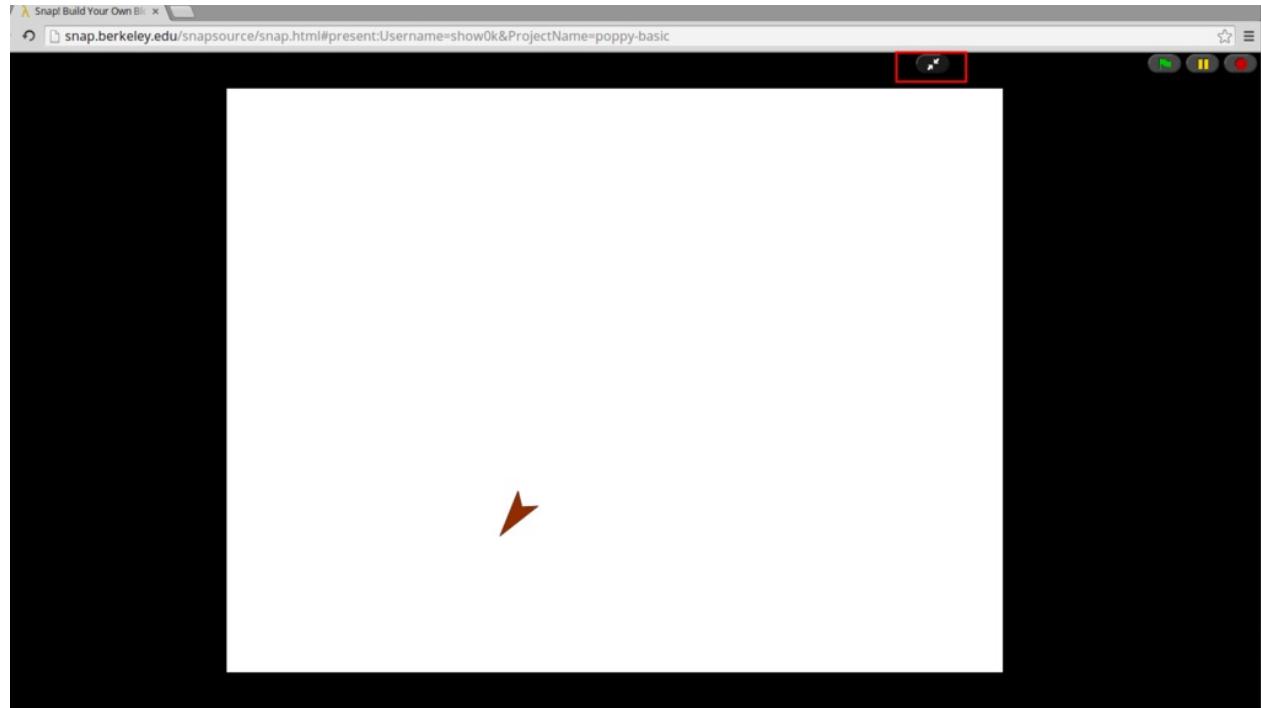


And this will re-open your project with the public sharing URL.



You can copy and paste the URL and share it by the way you want to your friends or to the Poppy community with the forum forum.poppy-project.org.

When you open a share project, the project is automatically opened in full screen on the sprite zone. To quit the full screen you have to click on the double arrow at the top of the snapshot below.



19.1.3.3 Export/Import your Snap! project

If you have a limited access to internet and you want to share project with other people, the best way is to export it:



A new tab in your web browser will be opened with an XML file like the picture below.

```

data:text/xml,<project name="super-poppy-program2" app="Snap! 4.0, http://snap.berkeley.edu" version="1">
<notes/>
<thumbnail>
  data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAKAAAAAB4CAYAAAB1ovlvAAADCULEQVR4Xu3VM5bARRE0c8+TG+W08GGAJwh1gMLceFIBENSTQeZSR03NrvPfn8K/vhdrvD18CI4E
</thumbnail>
<stage width="480" height="360" costume="0" tempo="60" threadsafe="false" lines="round" codify="false" inheritance="false" scheduled="false" id="1">
  <pentrails>
    data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAeAAAAFoCAYAAACPNyggAAA0hULEQVR4Xu3VwQkAAAjEMN1/abewn7jAQRC64wgQIECAAIF3gX1fNEiAAAECBAiMAHsCAgQIECA0C
  </pentrails>
  <costumes>
    <list id="2"/>
  </costumes>
  <sounds>
    <list id="3"/>
  </sounds>
  <variables/>
  <blocks/>
  <scripts/>
<sprites>
  <sprite name="Sprite" idx="1" x="0" y="0" heading="90">
    <costumes>
      <list id="9"/>
    </costumes>
    <sounds>
      <list id="10"/>
    </sounds>
    <variables/>
    <blocks/>
    <scripts>
      <script x="202" y="295">
        <block s="doGlide">
          <l>1</l>
          <l>0</l>
          <l>0</l>
        </block>
      </script>
      <script x="127" y="148">
        <block s="forward">
          <l>10</l>
        </block>
        <block s="setHeading">
          <l>90</l>
        </block>
      </scripts>
    </sprite>
  </sprites>

```

This file describes all your Snap! project in a simple file. It's not made to be human readable so don't be afraid, you just have to save it on your computer. For that, do a right click, choose "save as" and a proper name and location on your computer for this project.

If you want to import a previously exported project, you simply have to click on the import section of the file icon.



19.1.4 Search Poppy blocks

Every Poppy block in Snap! begins by a robot icon. So you can search all of them by the robot keyword. To search a specific block, do a right click on the block area, or use the keyboard shortcut CTRL+F.



19.1.5 First steps with Snap! and a Poppy creature

19.1.5.1 Test the connection between Snap! and your creature

19.1.5.1.1 If you are using a tangible robot

First, you must be connected to the same network LAN area than your robot (e.g. on the same router).

You have to go on the web homepage of your robot with its URL. You can use its IP address (for example <http://192.168.1.42>) if you have a way to know it or its hostname like <http://poppy.local>. To find its IP address you can see it in your router administration web page, or with a network scanner like "Fing" (for smartphone and desktop OS). To use directly its hostname <http://poppy.local> you must have a Zeroconf software installed on your computer, install "Bonjour print services for Windows" if you are running a Windows OS.

The home page of your poppy creature should look like the snapshot below:

- [Configure wifi and change hostname](#)
- [Open Ipython notebook](#): to control Poppy with Python code
- [Start Snap!](#), a block programmation webapp. It works faster with chromium/chrome browser
- [Start Poppy-monitor](#): a web app to easy start primitives and looking state of motors

- [Kill all acces to motors](#)
- [Display logs](#)

Click on the "Start Snap!" link to open the Snap! interface at start the connection with the Poppy robot.

Poppy special blocks are stored in the Examples. Go to "file" icon -> open -> Examples -> click on "Poppy blocks". It may take some time to load the blocks (~5-15 seconds), be patient.

19.1.5.1.2If you are using the V-REP simulator

Open V-REP, open a terminal (called *Command Prompt* on Windows), type and press Enter to execute the command below:

```
poppy-services --snap --vrep poppy-torso
```

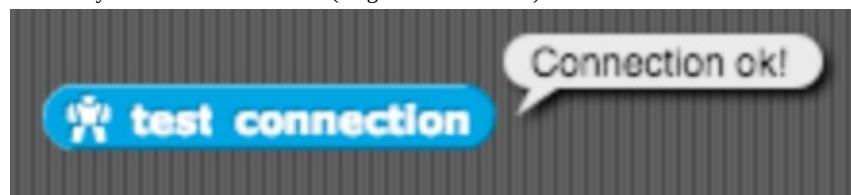
Substitute 'poppy-torso' with 'poppy-humanoid' or 'poppy-ergojr' to launch respectively a Poppy Humanoid or a Poppy Ergo Jr.

If you have issues, look at the section to see have more detailed informations.

After having executed this command a tab in your web browser should have been opened to Snap! have loaded Poppy blocks.

19.1.5.2Network

First test your connection with the (tangible or simulated) robot with the "test connection" block.



. if the block answer is "You may have connection troubles", your "host" variable inside the Snap! project is probably wrong. The host variable must be the IP or the hostname+.local" of your robot ; if you're using V-REP localhost is used to point to your own computer.



19.1.6Build your own blocks!

The functionality to build your own block was the first difference between Scratch and Snap! (now it's also possible to make custom blocks in Scratch)!

19.2Description of Poppy blocks

19.3Quick examples

19.3.1Record and play back a move

You can record a movement on one motor and play it back

You can change the speed

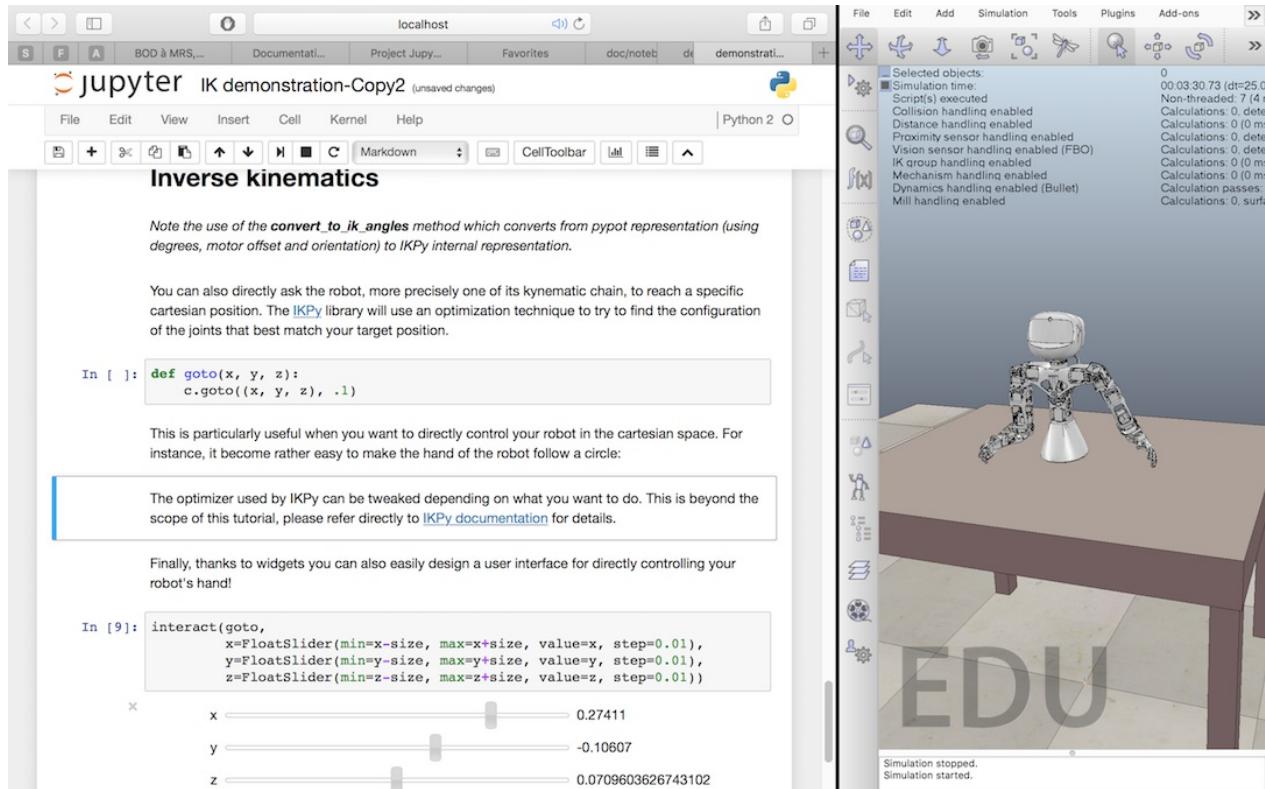
You can record it on all motors

or just a subset of your creature

19.3.1.1 Record and play demonstration movement

20 Jupyter Notebooks Gallery: using Python

Most of the existing examples of using Poppy robots in Python are given as [Jupyter Notebooks](#). We strongly encourage the use of this web application as it allows "the creation and sharing of documents that contain live code, visualization and explanatory text". Furthermore, they also permit the design of interface for live controlling a robot thanks to widgets.



This chapter presents a gallery of notebooks and tries to organize them into different categories:

- Getting started
- Simulator
- HTTP API and remote connection
- Scientific experiments
- Education
- Going further (advanced topics)

For each notebook, we provide a short description of what it does, with which robot/simulator it can be used and of course a link. Most of the notebooks are written in english but you will also find some in french (and hopefully soon in other languages).

If you see a notebook missing or want to submit your own, please [let us know!](#)

20.1 Getting started

- **Controlling a Poppy Humanoid in V-REP: [Notebook](#)** - Describe how to setup a Poppy Humanoid in V-REP and how to control it (motor control and sensor reading) from pyPop in Python.
- **Record, Save, and Play Moves: [Notebook](#)** - Simple introduction on how to record by demonstration moves on any Poppy Creature. It also shows how they can be re-played and saved/load to/from the disk.

20.1.0.1 Notebooks en français

- **10 choses à savoir avec Poppy Humanoid/ErgoJr et V-REP: [pour l'ErgoJr, pour l'Humanoid](#)** - 10 informations de base pour

bien commencer avec Poppy Humanoid ou Poppy ErgoJr simulés dans V-REP et comment les contrôler en Python.

20.2 Simulator

20.2.1 V-REP

- **Controlling a Poppy Humanoid in V-REP:** [Notebook](#) - Describe how to setup a Poppy Humanoid in V-REP and how to control it (motor control and sensor reading) from pyptot in Python.
- **Interacting with objects in V-REP:** [Notebook](#) - Show how you can programtically add objects to the V-REP simulation and interact with them. This example uses a Poppy Torso but can be easily adapted to other creatures.
- **V-REP simulation benchmark:** [Notebook](#) - Check how fast the V-REP simulation is running on your computer. Can be used with the **Humanoid, Torso, and ErgoJr**.
- **Learning the robot IK:** [Notebook](#) - Demonstrate how explauto can be used to learn the inverse kinematics of a Poppy robots. The experiments are run in V-REP simulation but it also gives hints on how it can be transposed in the real world.

20.2.1.1 Notebooks en français

- **10 choses à savoir avec Poppy Humanoid/ErgoJr et V-REP** [Notebook pour l'ErgoJr](#)[Notebook pour l'Humanoid](#) - 10 informations de base pour bien commencer avec Poppy Humanoid ou Poppy ErgoJr simulés dans V-REP et comment les contrôler en Python.

20.3 HTTP REST API and remote connection

- **Controlling a robot using HTTP requests:** [Notebook](#) - Show how you can send HTTP requests to a robot, using the REST API, to control it. The notebook is based on a V-REP simulated Poppy Humanoid but can be adapted to other creatures.

20.4 Scientific experiments

20.4.1 Discover Explauto

- **Learning the robot IK:** [Notebook](#) - Demonstrate how explauto can be used to learn the inverse kinematics of a Poppy robots. The experiments are run in V-REP simulation but it also gives hints on how it can be transposed in the real world.

20.5 Demo interface

- **Primitives launcher:** [Notebook](#) - Provides all codes needed to directly launched primitives (stand, sit, idle motions, limit torque...)
- **Simple demo - livecoding:** [Notebook](#) - Show some basic examples of what you can do with a Poppy Humanoid: start primitives, control motors, get sensor readings. It was designed for a 15 minutes live coding session (in French).

20.6 Education

20.6.0.1 Notebooks en français

20.6.0.2 Initiation à l'informatique en Lycée

- **Découverte:** [TP1](#), [TP2](#), [TP3](#) - Comprendre comment faire bouger simplement le robot. Utilisation des boucles. Ces TPs utilisent un Poppy Torso simulé dans V-REP.

- **Dialogue:** [TP1](#), [TP2](#) - Établir un dialogue entre Python et le robot. Ces TPs utilisent un Poppy Torso simulé dans V-REP.

20.7 Going further

20.7.1 Low-level communication

- **Configure a new motor directly in Python:** [Notebook](#) -

20.7.2 Debug and setup

- **Configure a new motor directly in Python:** [Notebook](#) -

20.7.3 Benchmark

- **Trajectory following:** [Notebook](#) - Measure how precisely a **Poppy mini-4dof** follows a given trajectory. It measures the position, speed and load of multiple motors on a fast and a slow trajectory.
- **V-REP simulation benchmark:** [Notebook](#) - Check how fast the V-REP simulation is running on your computer. Can be used with the **Humanoid**, **Torso**, and **Ergo Jr.**

20.7.4 Extending Poppy software

- **Define your own Poppy creature:** [Notebook](#) -

20.8 Contributing to this gallery

Do not hesitate to let us know if some cool Notebooks are missing! You can directly send pull-request on GitHub or uses the [issue tracker](#).

21 Programming Poppy robots in Python

Make your Ergo Jr jump in 3 lines of code

```
In [ ]: from poppy.creatures import PoppyErgoJr
         jr = PoppyErgoJr()

In [ ]: jr.jump.start()
```

This chapter will guide you through how to control Poppy robots in Python. As it is actually the language used for writing Poppy core libraries, you will see how to access all the different levels of control, from the higher to the lower.

We will detail everything you need to know to directly program your robot using the Python embedded in the Poppy robot or to install everything locally. Note that this chapter does not intend to teach you Python or programming from scratch and thus if you are completely new to Python it may be good to start with a Python tutorial. Yet, we try to keep the tutorials as simple as possible and we will always warn you when some parts are targeting more advanced users.

We will try to provide as many examples as possible and point to the complete API so you can find and use the least famous features. Most of the examples and tutorials are available as a collection of [Jupyter notebooks](#). The next chapter, [Jupyter Notebooks Gallery](#), presents a list describing each notebook, what they will teach, what they can be used for, for which robot, etc.

All Poppy libraries are open source and are released under the [GPL v3](#) license. So you can freely access the source code on [github](#). Do not hesitate to fork them, send pull request and contribute!

21.1 Why Python and Anaconda?



The libraries developed for the Poppy project were designed with the aim to make it easy and fast to write code for controlling various robots based on - originally - robotis dynamixel motors. The idea was to provide access from the lower level - raw serial communication with a specific motor for instance - to higher levels such as starting and stopping primitives/behaviors (e.g. face tracking, postures, ...) or directly recording motions through learning by demonstration.

We decided to write most of them in Python as its flexibility allows for fast and modular development. It was also meant to be accessible by a large audience, from developers and roboticists in general, to hobbyists, researchers, artists... Python was also chosen for the tremendous pools of existing libraries (scientific, computer vision, IO, web...) so if one is interested in adding a new feature, such as support for a new motor/sensor, it should be as easy and fast as possible.

Finally, support for multiplatforms and ease of installation were also key aspects.



Anaconda

We also strongly advise to use the [Anaconda Python distribution](#) as it already includes most of the libraries needed by the Poppy libraries. We also provide all poppy libraries as conda recipes so they can be easily install using Anaconda (see the [install section](#)).

21.2 Overview of the different libraries

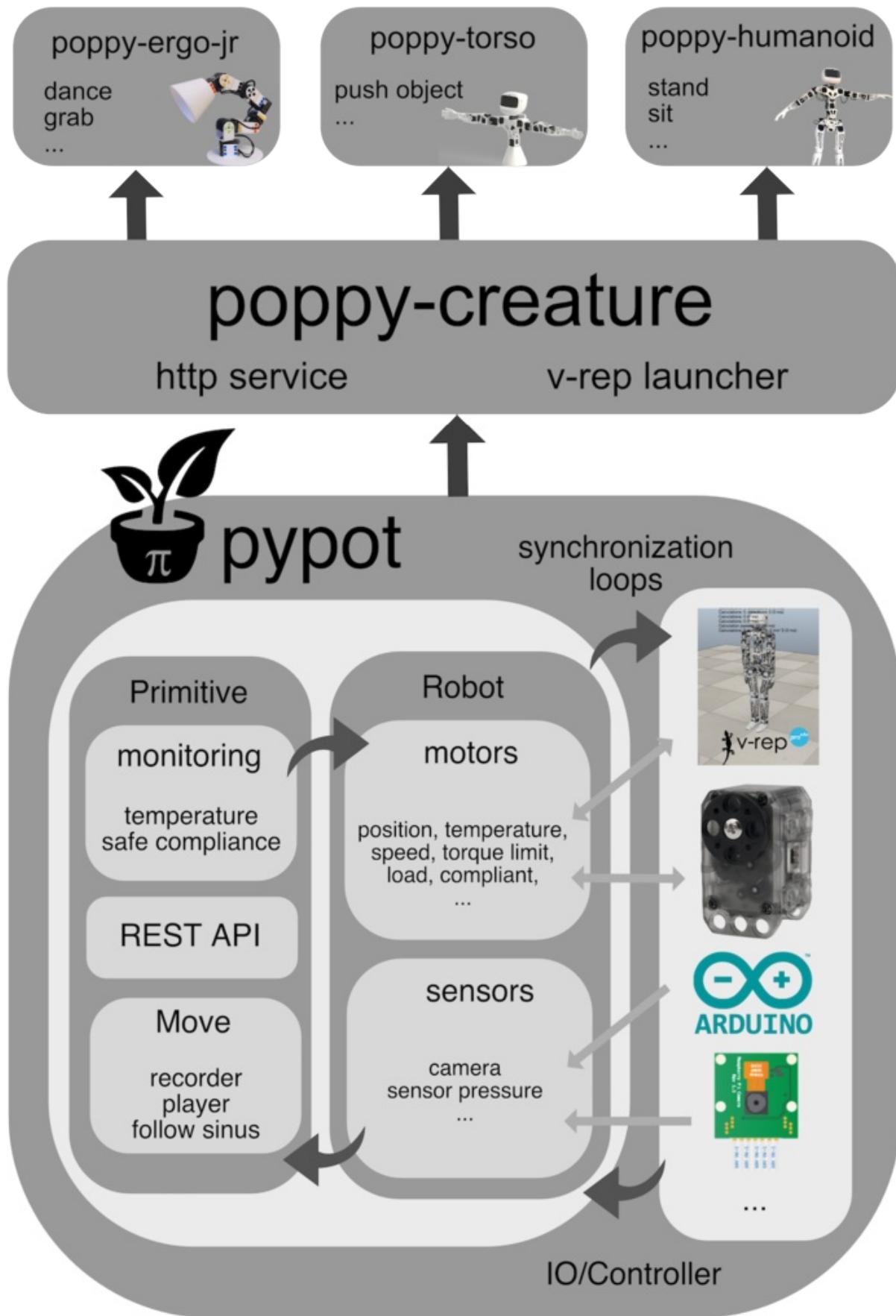
A more detailed documentation of these software libraries is available in the [software libraries section](#)

Before jumping into the code, we will briefly introduce the different existing Poppy libraries and how they interact with each other.

There are three main library levels:

- [pypot](#): This is the core of the Poppy software architecture. Pypot handles all the low level communication with the hardware (both sensors and motors), defines synchronization loops so your command are always up to date. It also provides the primitives mechanism which allows the definition of simple behavior that can be - more or less - automatically combined.
- [poppy-creature](#): This library defines the common tools shared by all Poppy robots, for instance how to automatically launch the simulator or start the HTTP API attached to any robot.
- [poppy-ergo-jr](#), [poppy-torso](#), and [poppy-humanoid](#): Those libraries are specific to their respective Poppy robot. They define the particular configuration of the robot, the sensors it uses, which motors are connected to which buses... This is also where behaviors specific to a creature are defined (the stand primitive for the humanoid for instance).

This is summarized in the diagram below:



21.3 Installation

First, note that if you are only planning to use real robots, they already come with Python and all Poppy libraries installed. You can directly connect to the Jupyter notebook server via the [web interface](#) and have nothing to install on your machine!

What you need to install is summarized in the diagram below:



Yet, if you are planning to either

- Use a simulator (e.g. V-REP, or web simulator),
- or want to directly plug the robot to your computer

You will have to install Poppy libraries locally. They work on Windows, Mac OS, Linux, and have been tested on:

- Python >= 2.7
- Python >= 3.4

Also note that if you are planning to directly plug your robot to your USB port, specific drivers should be installed.

All steps are detailed in the chapter [Manual installation](#).

21.4 Quickstart: Hello Poppy world!

To give you a rapid overview of what you can do using Python to program Poppy robots, this section will show you how to:

- Create and connect your robot
- Retrieve values from the sensor and send motor commands
- Start playing with primitive by recording motions by demonstration

This section does not intend to cover everything that can be done in Python with Poppy but to give you sneak peaks of the most common features. For more advanced use, you should refer to the next section where we present a list of Jupyter notebooks each detailing a specific aspect or feature.

In the following examples, we assume that you have a working environment meaning that you either:

- are using the Python embedded in your robot: through the Jupyter Notebook server,
- or you have installed everything locally to work with a simulator.

21.4.1 Create and connect to a Poppy robot

21.4.1.1 Import the library

The very first step you have to do to start programming Poppy robots in Python is to import the library. In Python they are called [module or package](#).

To do that, you write something similar to:

```
from poppy.creatures import *
```

This will actually import all Poppy robots installed on the Python distribution you are using. If you want to use a specific robot, you can replace the * (which means all here) by the name of the robot you want.

For the ErgoJr:

```
from poppy.creatures import PoppyErgoJr
```

For the Torso:

```
from poppy.creatures import PoppyTorso
```

For the Humanoid:

```
from poppy.creatures import PoppyHumanoid
```

If you see an error similar to the one below when executing the previous line, this means that the libraries are not correctly installed. See the section [TODO](#).

```
In [1]: from poppy.creatures import PoppyHumanoid
-----
ImportError                               Traceback (most recent call last)
<ipython-input-1-18e4c5a36525> in <module>()
----> 1 from poppy.creatures import PoppyHumanoid

ImportError: cannot import name PoppyHumanoid
```

21.4.1.2 Create the Robot object - with a real robot

Then, you can actually create the Python object that will represent your robot. Depending on the Poppy robot you are using:

```
# if you are using an Ergo Jr
poppy = PoppyErgoJr()
```

or

```
# if you are using a Torso
poppy = PoppyTorso()
```

or

```
# if you are using a Humanoid
poppy = PoppyHumanoid()
```

And that's it, if you did not see any error message it means that you are connected to your robot. If you see an exception like the one shown below, you should check the wire connection and try again:

```
IOError: Connection to the robot failed! No suitable port found for ids [3, 5, 7, 11, 13, 17]. These ids are missing [3]
```

21.4.1.3 Create the Robot object - with V-REP

To use a simulated robot instead of a real one, you only have to specify it when creating the Robot object. For instance, if you want to create a simulated Poppy Torso, you simply have to execute the following line:

```
poppy = PoppyTorso(simulated='vrep')
```

All three Poppy robots - Humanoid, Torso, and Ergo Jr - can be used with V-REP.

If you see an error message like this, check that you have launched V-REP and that you have close the popup in V-REP (see #TODO for details).

```
IOError: Connection to V-REP failed!
```

21.4.1.4 Create the Robot object - with web simulator

Currently only the Ergo Jr is usable within the web simulator.

It also requires specific versions of libraries to be used properly.

To make sure you meet these requirements, you can type this command from your shell:

```
pip install pypot>=2.12 poppy-creature>=1.8 poppy-ergo-jr>=1.6 --upgrade
```

You can then instantiate the poppy-ergo-jr creature:

```
poppy-services --poppy-simu --snap --no-browser poppy-ergo-jr
```

This will create a server for Snap! on port 6969, and a server for the visualizer on port 8080.

You can then head to the [visualizer page](#).

21.4.2 Access the sensors and motors

The robot object you just created contains two main groups of objects:

- motors
- sensors

that can be easily access using `poppy.motors` and `poppy.sensors`. As soon as the robot object is created it automatically starts synchronization loops which will ensure that the last available value are received/sent to the robot.

Servomotors that are used in Poppy robots can be seen as both motors and sensors. Indeed, on top of being "simple" motors, they also provide multiple sensing information: their current position, speed and load but also their temperature, the current used... Yet, for simplification they are only available under the motor category.

21.4.2.1 Get data from your robot

Now that you have created your robot object, you can directly use Python to discover which motors are attached.

In all examples below the results are shown for an ErgoJr. If you are using a Torso or a Humanoid you will see more motors with different names.

For instance, to know how many motors your robot have you can execute:

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

poppy.motors is actually a list of all motors connected to your robot. Thus, if you want to get the present position of all motors, you can do:

```
for m in poppy.motors:
    print(m.present_position)
```

Of course, you can also access a specific motor. To do that, you need to know the name for the motor you want to access. You can find this list in the assembly documentation of your robot.

You can also get a list of all motors name directly from python:

```
for m in poppy.motors:
    print(m.name)
```

or using a motor pythonic expression:

```
print([m.name for m in poppy.motors])
```

Then you can directly access the desired motor by its name:

```
m = poppy.m3
```

or get its position:

```
print(poppy.m3.present_position)
```

The most common values for motors are:

- `present_position`
- `present_speed`
- `present_load`

Similarly, you can get data from your sensors. Depending on the Poppy robot you have different sensors available. You can get the list of all sensors in the exact same way you did for motors:

```
print([s.name for s in poppy.sensors])
```

And then access a specific sensors by its name. For instance, to get an image from the camera of the Ergo Jr:

```
img = poppy.camera.frame
```

This section just presented some of the available values that you can get from your motors/sensors. There are many other - some are specific to a particular robot - we will present them through the different notebooks.

21.4.2 Send motor commands

Now that we have shown you how to read values from your robot, it is time to learn how to make it move!

This is actually really similar to what you have just seen. Instead of getting the *present_position* of a motor you simply have to set its *goal_position*.

But first, you have to make sure your motor is stiff, meaning that you cannot move it by hand. To do that we will turn off its compliancy. Assuming you have an Ergo Jr and want to make the motor *m3* moves - feel free to use any other motor but make sure the motor can freely move without hurting any of your finger:

```
poppy.m3.compliant = False
```

The motor should now be stiff. And then, to make it move to its zero position:

```
poppy.m3.goal_position = 0
```

Note: *present_position* and *goal_position* are actually two different registers. The first refers to the current position of the motor (read only) while the second corresponds to the target position you want your robot to reach. Thus, they can have different values while the motor is still moving to reach its *goal_position*.

As a slightly more complex example we will make it go to 30 degrees then -30° three times:

```
import time

for _ in range(3):
    poppy.m3.goal_position = 30
    time.sleep(0.5)
    poppy.m3.goal_position = -30
    time.sleep(0.5)
```

Note that after each new value set to *goal_position* we wait so the motor has enough time to actually reach this new position. Another way to do the same thing is to use the *goto_position* method:

```
import time

for _ in range(3):
    poppy.m3.goto_position(30, 0.5, wait=True)
    poppy.m3.goto_position(-30, 0.5, wait=True)
```

As you can see, this method takes three arguments, the target position, the duration of the move and whether to wait or not the end of the motion.

If you want to move multiple motors at the same time, you can simply do something like:

```
for _ in range(3):
    poppy.m1.goal_position = -20
    poppy.m3.goal_position = 30
    time.sleep(0.5)
    poppy.m1.goal_position = 20
    poppy.m3.goal_position = -30
    time.sleep(0.5)
```

or use a python dictionary storing the target position per motor you want to move, that can be given to the *goto_position* method:

```

pos_1 = {'m1': -20, 'm3': 30}
pos_2 = {'m1': 20, 'm3': -30}

for _ in range(3):
    poppy.goto_position(pos_1, 0.5, wait=True)
    poppy.goto_position(pos_2, 0.5, wait=True)

```

You can turn a motor back to its compliant mode (where you can freely move it) by setting its compliant register to True:

```
poppy.m3.compliant = True
```

21.4.3 Record and play motion by demonstration using primitives

Pypot provides you with the primitive mechanism, which are simply pre-defined behaviors that can be attached to your robot. In this section, we will show you how to use some primitives already existing for recording and playing motions. You can also define your own primitive but this is out of the scope of this section, you will find details on how to do this in dedicated notebooks.

21.4.3.1 Record a motion by demonstration

Designing choreographies for your robot using `goal_position` or `goto_position` can be long and kind of troublesome. Fortunately, there is a much more efficient way of doing this: recording motions by directly demonstrating the move on the robot.

This can be summarized into few steps:

- make the robot compliant so you can move it by hand
- start the recording
- actually moves the robot so it follows whatever move/choreography you can think of
- stop the recording

And now to do that in Python:

So, first we turn all motors of the robot compliant:

```

for m in poppy.motors:
    m.compliant = True

```

You can also record a movement with motors stiff (`compliant = False`), and moving them with `goal_position` or `goto_position` commands.

Then, we have to include the primitive used for recording motion:

```
from pypot.primitive.move import MoveRecorder
```

To create this primitive, you have to give the following arguments:

- on which robot you want to use this primitive (this can be useful if you are working with multiple robot at a time - for instance you can record a move on a robot and at the same time make it reproduce by another one: this [notebook](#) will guide you on how to do this).
- the record frequency of the move you want to register: how many position per second will be recorded - the higher the more accurate the record will be but also more data will have to be processed - good values are usually between 10Hz and 50Hz.
- the motors that you want to record. you can record a move on a subpart of your robot, for instance only on the left arm.

Here, we will record a move on the whole robot at 50Hz:

```
recorder = MoveRecorder(poppy, 50, poppy.motors)
```

We used `poppy.motors` to specify that we want all motors if you only want let's say the two first motors of an Ergo Jr you could have used `[poppy.m1, poppy.m2]` instead.

Now it is time to record. As it can be hard to both move the robot and type Python command at the same time, we will make a small script, that:

- wait 5s so you can get ready to record
- start the record
- record for 10 seconds
- stop the records

```
import time

# Give you time to get ready
print('Get ready to record a move...')
time.sleep(5)

# Start the record
record.start()
print('Now recording !')

# Wait for 10s so you can record what you want
time.sleep(10)

# Stop the record
print('The record is over!')
record.stop()
```

Now, you should have a move recorded. You can retrieve it from the recorder primitive:

```
my_recorded_move = record.move
```

and check how many positions where recorded:

```
print(len(my_recorded_move.positions()))
```

21.4.3.2 Replay recorded moves

Now to play back recorded motions you have to use another primitive: MovePlayer

```
from pypot.primitive.move import MovePlayer

player = MovePlayer(poppy, my_recorded_move)
```

As you can see, to create it you have to specify the robot (as for the MoveRecorder) and the move you want to play.

Automatically all recorded motors become stiff to be able to play the move.

Then, you can simply start the replay:

```
player.start()
```

And if you want to play it three times in a row:

```
for _ in range(3):
    player.start()
    player.wait_to_stop()
```

We use the `wait_to_stop` method to make sure we wait for the first move to finish before we start another. By default, playing a move we will not block to allow you to play multiple move in parallel.

21.4.4 Write a simple sensori-motor loop

Robotic is all about sensori-motor loops, meaning that motor commands will be more or less directly related to the sensor readings. In other terms the robot actions will be determined by what it perceives from its environment.

Poppy libraries and more particularly pypot provides you with tools to easily write sensori-motor loops. We will show here a very simple example where some motor of an Ergo Jr will be controlled by the position of other motors in order to keep the head of the Ergo Jr straight.

To do that, we will free the two first motors, so they can be moved by hand. Two other motors will try to lively compensate the motion applied on the free motors.

We need few simple steps:

1. read values from sensors (here the two free motors)
2. compute command from those readings
3. set new motor command
4. go back to step 1.

This example is designed for the Ergo Jr. It could be adapted to other Poppy robots, by changing the motors used. Yet, it is not that obvious which one to use to have a "cool" result.

21.4.4.1 Demo version

Before writing the sensori-motor loop, we will first set the Ergo Jr in a base position.

```
from poppy.creatures import PoppyErgoJr

jr = PoppyErgoJr()

jr.goto_position({'m1': 0.,
                  'm2': -60.,
                  'm3': 55.,
                  'm4': 0.,
                  'm5': -55.,
                  'm6': 60.}, 2., wait=True)
```

Then, we make sure the *moving speed* of the motors are not too high to prevent shaky motions:

```
for m in jr.motors:
    m.moving_speed = 250
```

Finally, we free the two first motors:

```
jr.m1.compliant = True
jr.m2.compliant = True
```

Now, that everything is setup we write our very simple sensori-motor loop like this:

```

import time

while True:
    # Step 1
    p1 = jr.m1.present_position
    p2 = jr.m2.present_position

    # Step 2
    g1 = -p1
    g2 = -p2

    # Step 3
    jr.m4.goal_position = g1
    jr.m6.goal_position = g2

    time.sleep(.02)

```

- **Step 1:** As you can see, here our readings step is simply to retrieve the *present_position* of the motors *m1* and *m2*.
- **Step 2:** Here, we defined the base position so the motors *m1/m4* and *m2/m6* are parallel. Thus, to compensate the head position, we simply have to define the new motor goal position as the opposite of the read present position.
- **Step 3:** We simply set the goal position as the just computed command

Those steps are included inside an infinite loop - with a `time.sleep` to avoid CPU overhead.

To stop this *while True* loop, you will have to use the classical Ctrl-c, or use the stop button if you are running it through Jupyter.

21.4.4.2 Now with a primitive

But what about if you want to make this behavior an independent "brick" that you can start/stop on demand combine with other behaviors. Well, primitives are meant to do just that.

There are two main types of primitive: *Primitive* and *LoopPrimitive*. The first one basically gives you access to just a *run* method where you can do everything you want on a robot. The second one as the name indicates is an infinite loop which calls an *update* method at a pre-defined frequency. In our case it is the more suited one.

Here is the entire definition of this primitive:

```

class KeepYourHeadStraight(LoopPrimitive):
    def setup(self):
        for m in self.robot.motors:
            m.compliant = False

        self.robot.goto_position({'m1': 0.,
                                'm2': -60.,
                                'm3': 55.,
                                'm4': 0.,
                                'm5': -55.,
                                'm6': 60.}, 2., wait=True)

        for m in self.robot.motors:
            m.moving_speed = 250

        self.robot.m1.compliant = True
        self.robot.m2.compliant = True

    def update(self):
        self.robot.m4.goal_position = -self.robot.m1.present_position
        self.robot.m6.goal_position = -self.robot.m2.present_position

```

As you can see, there are two main parts. The *setup* method which defines what needs to be done to prepare the robot before starting the behavior - here simply puts it in its base position and turns on the compliance for the two first motors.

And the *update* method which will be regularly called: here is where we put the actual code for the sensori-motor loop: reading sensor - computing the new command - and sending the new command to the motors.

Now that we have defined our primitive, we can instantiate it and start it:

```
# we specify we want the primitive to apply on the jr robot instance
# and that the update method should be called at 50Hz
head_straight = KeepYourHeadStraight(jr, 50.0)

head_straight.start()
```

You can stop it whenever you want:

```
head_straight.stop()
```

And re-starting it again...

```
head_straight.start()
```

The huge advantage of using a primitive in this case is that after starting it, you can still easily run any other codes that you want. The primitive starts its own thread and thus runs in background without blocking the execution of the rest of the code.

22Use the REST API to control a Poppy Robot

23**Gallery of activities**

23.1Contrôler Poppy avec un Arduino via Snap4Arduino

Written by [Gilles Lassus](#).

Objectif : contrôler un moteur de Poppy par un potentiomètre sur la platine Arduino.

23.0.1Préparation de Snap4Arduino

- Téléchargez et installez [Snap4Arduino](#).
- Téléchargez les blocs [pypot-snap-blocks.xml](#). Ils devront être importés dans Snap4Arduino à chaque démarrage.

23.0.2Préparation de l'Arduino

- Connectez votre platine, ouvrez Arduino et téléversez le firmware StandardFirmata. (disponible via Fichier - Exemples - Firmata).
- Branchez un potentiomètre sur la sortie analogique A0, comme illustré ci-dessous :



23.0.3Lancement de la simulation (dans le cas d'un Poppy simulé dans Vrep)

- Lancez Vrep .
- Exécutez les commandes python suivantes :

```
from poppy.creatures import PoppyHumanoid

poppy = PoppyHumanoid(simulator='vrep', use_snap=True)
```

puis

```
poppy.snap.run()
```

23.0.4Ouverture de Snap4Arduino

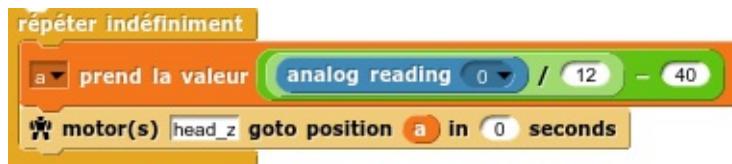
- Lancez Snap4Arduino et importez les blocs [pypot-snap-blocks.xml](#). (une fenêtre avertissant que le projet importé a été créé par Snap! apparaît ; elle est sans conséquence.)
- Dans les blocs Arduino, cliquez sur *Connect Arduino* pour établir la connexion entre Snap4Arduino et votre platine.



Un message de confirmation apparaît, signe que la connexion est effective.

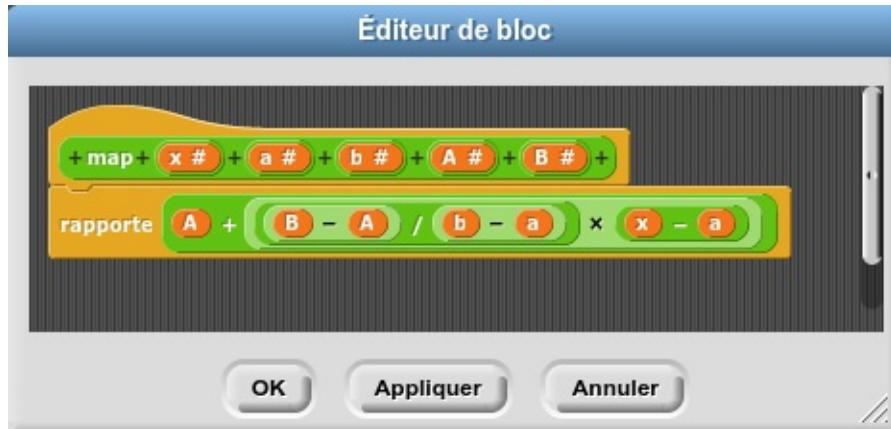
23.0.5Commander un moteur via le potentiomètre

La valeur analogique lire dans A0 est un entier entre 0 et 1024. Pour la "mapper" entre (environ) -40 et 40, on la divise par 12 avant de lui soustraire 40. On peut donc alors construire l'instruction suivante, qui fera bouger le moteur *head_z* de Poppy entre -40° et +40° :

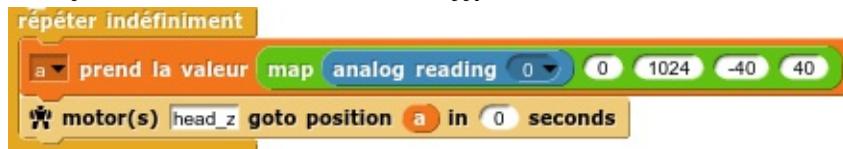


23.0.6 Remarques diverses

- Il peut être utile de créer un bloc *map* (une version [ici](#)) équivalent à la fonction éponyme d'Arduino, permettant de mettre à l'échelle automatiquement une valeur dans une plage donnée :



Le script de commande du moteur *head_z* de Poppy entre -40° et 40° deviendrait alors :



Cette méthode de contrôle a pour principal défaut de "bloquer" la carte Arduino avec le StandardFirmata : il serait plus agréable de pouvoir simplement lire les données du port série envoyées par l'Arduino, et ainsi pouvoir téléverser le programme de son choix dans l'Arduino. Ceci est discuté [ici](#). Toutefois, la page du projet [Snap4Arduino](#) liste les composants annexes (LCD display, UltraSound Sensor) pouvant être directement contrôlés, et explique en [détail](#) comment modifier le StandardFirmata pour intégrer un nouveau composant.

25 Installation for advanced users

You may want to install Poppy software only if you are in one of these situations:

1. You want to control a simulated robot.
2. [special case for advanced users] You want to install yourself the operating system of your robot instead of using a [pre-made ISO image](#).
3. [special case for advanced users] You want to control a Poppy creature from your computer **without** using the Raspberry Pi board.

Note: Though first situation is a "standard case", the second and third ones are affordable only by advanced users who have a good comprehension of the global system.

25.1 Install the Python Interpreter and Poppy software

Poppy is run by Python computer code. Depending on your operating system you will have to install Python and in any case you'll have to install the required software libraries.

Whatever your operating system if you are getting started with Python and want to install a full Python environment for scientific computing, **we suggest you to use [Anaconda Python distribution](#)**.

25.1.1 Install everything needed for a Poppy Board

The easiest way to setup the control board of your Poppy Creature is to use one of the ready-made SD-card images.

They come with everything installed and ready, you just need to copy it on a SD-card and you are good to go. For that, you will need a free 8Go (or more) SD-card, and download the image corresponding to your board and write it to your SD-card.

The [startup section](#) describes this procedure.

Download the image of your system:

- [Raspbian Jessie](#) if you are using a **Raspberry Pi 2**.
- [A special Ubuntu 14.04](#) if you are using a Odroid U3.

Write the image to the SD-card with you favourite disk writer tool as explained in the [startup section](#).

Connect to your board in SSH:

- For the Raspberry Pi: `ssh pi@raspberrypi.local , password=raspberry`.
- For the Odroid U3: `ssh odroid@odroid.local , password=odroid`.

Download the install script:

```
wget https://raw.githubusercontent.com/poppy-project/poppy-installer/master/poppy-configure.sh
```

This script takes two parameters:

- The board name {odroid, rpi}.
- The creature name {poppy-humanoid, poppy-torso, poppy-ergo-jr}.

To build the system for a Poppy Ergo Jr on a Raspberry Pi, one would run:

```
bash poppy-configure.sh rpi poppy-ergo-jr
```

Reboot after the end of the installation.

The hostname, default user and password will be all set to "poppy" (`ssh poppy@poppy.local password=poppy`).

You can test your installation with the web interface in your web browser <http://poppy.local>.

If there is any issue don't hesitate to post a message on the [issue tracker on GitHub](#) or in the dedicated section of [the forum](#).

25.1.2 Install Python and Poppy software on Windows

If you want a step by step screencast of the installation of Anaconda and V-REP on Windows, you can see [these videos](#).

25.1.2.1 Install Python

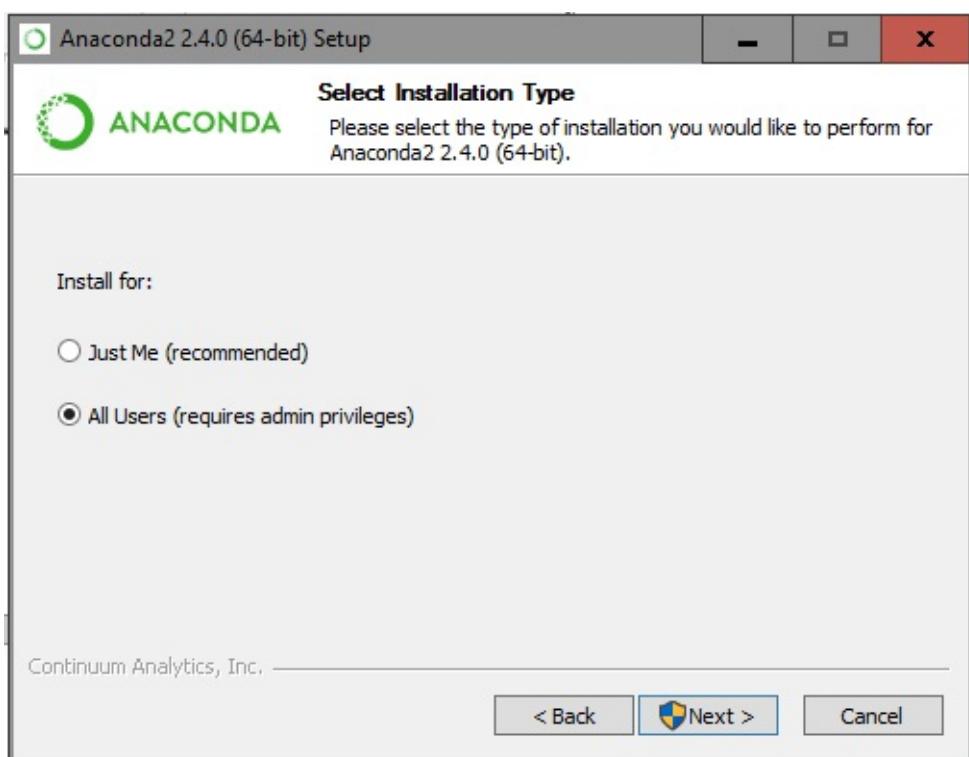
We encourage the use of the Anaconda Python distribution.

However, if you already installed a Python distribution like Canopy (shipped with scientific packages), you can directly [install Poppy software](#).

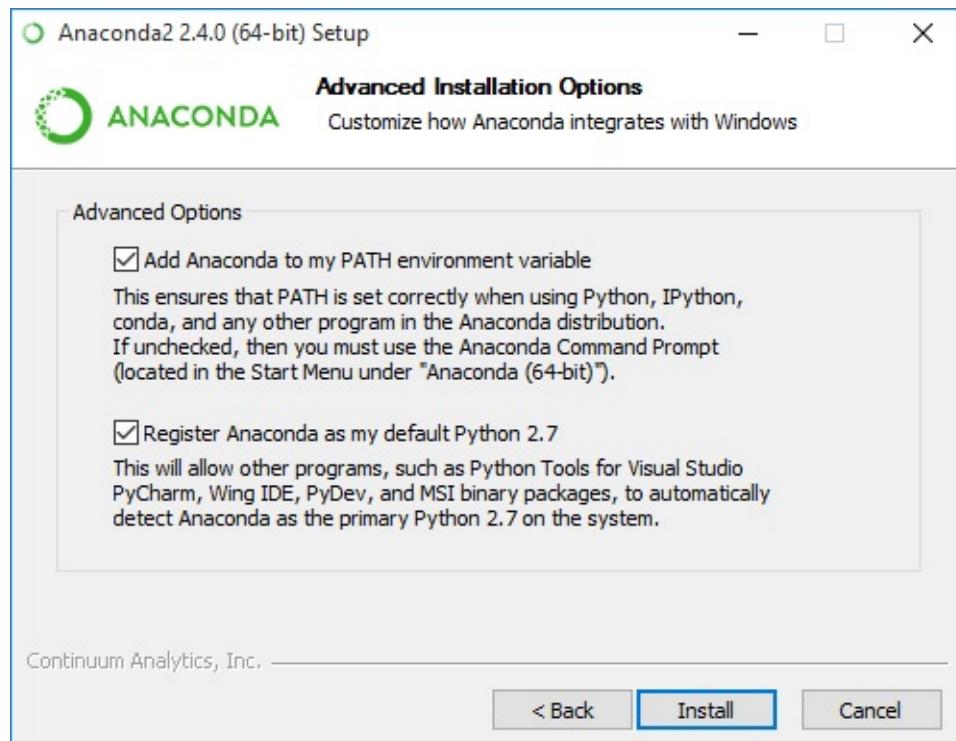
25.1.2.1.1 Anaconda

Download Anaconda Python distribution (400 MB) [here for 64-bit computer](#) or [here for 32-bit](#).

Install it by clicking on "next" at each step. If you intend to install Anaconda for all users of your computer, be sure to select "all users".



It is also very important that the two check-boxes of the PATH and the default Python are checked.



Now you have a Python distribution ready to [install Poppy software](#).

25.1.2.1.2Miniconda (alternative to Anaconda)

Miniconda is a "light" version of Anaconda which contain only Python and the conda package manager. You can install it **instead of Anaconda** and save a lot of disk space (25 Mo vs 400 Mo), but you will have to do another step in the install process.

Download miniconda [here for 64-bit](#) computer or [here for 32-bit](#) computer.

Install it and be sure that the two check-boxes of the PATH and the default Python are checked.

Open the Command Prompt (press the windows key and type "Command Prompt"), type and press Enter to execute the command below:

```
conda install numpy scipy ipython-notebook matplotlib
```

Now you have a Python distribution ready to [install Poppy software](#).

25.1.2.2Install Poppy software

Open the prompt of your Python Distribution (called *Anaconda Prompt* for Anaconda) or the *Command Prompt* of Windows, type and press Enter to execute the command below:

```
[Anaconda2] C:\Users\Luc>pip install poppy-torso
Activating environment "C:\Anaconda2"...
Collecting poppy-torso
  Downloading poppy_torso-1.1.4-py2-none-any.whl (1.3MB)
    100% [#####| #####] 1.3MB 242kB/s
Requirement already satisfied (use --upgrade to upgrade): pyplot>=2.7.0 in c:\anaconda2\lib\site-packages (from poppy-torso)
Requirement already satisfied (use --upgrade to upgrade): poppy-creature>=1.6.0 in c:\anaconda2\lib\site-packages (from poppy-torso)
Requirement already satisfied (use --upgrade to upgrade): scipy in c:\anaconda2\lib\site-packages (from pyplot>=2.7.0->poppy-torso)
Requirement already satisfied (use --upgrade to upgrade): enum34 in c:\anaconda2\lib\site-packages (from pyplot>=2.7.0->poppy-torso)
Requirement already satisfied (use --upgrade to upgrade): pyserial>2.6 in c:\anaconda2\lib\site-packages (from pyplot>=2.7.0->poppy-torso)
Requirement already satisfied (use --upgrade to upgrade): numpy in c:\anaconda2\lib\site-packages (from pyplot>=2.7.0->poppy-torso)
Requirement already satisfied (use --upgrade to upgrade): bottle in c:\anaconda2\lib\site-packages (from poppy-creature>=1.6.0->poppy-torso)
Installing collected packages: poppy-torso
Successfully installed poppy-torso-1.1.4

[Anaconda2] C:\Users\Luc>
```

```
pip install poppy-torso --user -U --no-deps
```

This will install everything necessary to control a Poppy Humanoid. Substitute "poppy-torso" with "poppy-humanoid" or "poppy-ergojr" to install respectively a Poppy Humanoid or a Poppy Ergo Jr.

In case of update, it is advised to upgrade Pypot (the motor library control) and the creature package separately:

```
pip install pypot --user -U --no-deps
pip install poppy-torso --user -U --no-deps
```

25.1.3Install Python and Poppy software on Mac OSX

Mac OSX has a Python distribution installed by default. Before installing Poppy software, you need to install the Python package manager pip.

Open a terminal and execute the command below: `curl --silent --show-error --retry 5 https://bootstrap.pypa.io/get-pip.py | sudo python`

You can now install Poppy software for the creature of your choice: `pip install poppy-torso --user -U --no-deps`

Substitute "poppy-torso" with "poppy-humanoid" or "poppy-ergojr" to install respectively a Poppy Humanoid or a Poppy Ergo Jr.

25.1.4Install Python and Poppy software on GNU/Linux

Most of GNU/Linux distributions, have already a Python distribution installed by default.

25.1.4.1Using the default Python distribution

Pypot, the main library of the robot is depending (amongst some other) on two big scientific libraries *Numpy* and *Scipy* which are themselves depending on C and Fortran code. These libraries may be installed with the Python package system (pip), but because of the huge number and differences between GNU/Linux distributions pip is not able to distribute binaries for Linux so all dependencies must be compiled... The solution to avoid the compilation of numpy and scipy is to install them with your distribution package manager.

On Ubuntu & Debian:

```
curl --silent --show-error --retry 5 https://bootstrap.pypa.io/get-pip.py | sudo python
sudo apt-get install python-numpy python-scipy python-matplotlib python-dev
```

On Fedora:

```
curl --silent --show-error --retry 5 https://bootstrap.pypa.io/get-pip.py | sudo python
sudo yum install numpy scipy python-matplotlib
```

On Arch Linux:

```
curl --silent --show-error --retry 5 https://bootstrap.pypa.io/get-pip.py | sudo python
sudo pacman -S python2-scipy python2-numpy python2-matplotlib
```

You can now [install Poppy software](#).

Note: The downside is the Python libraries from you distribution system are very often out of date.

25.1.4.2Using Anaconda (or miniconda)

If you want to have up to date numpy, scipy and ipython without having to compile them, we suggest you to install Anaconda or at least the conda package manager distributed with miniconda. Download miniconda (64-bit) with these command below in your terminal:

```
curl -o miniconda.sh http://repo.continuum.io/miniconda/Miniconda-latest-Linux-x86_64.sh If you have a 32-bit computer
curl -o miniconda.sh http://repo.continuum.io/miniconda/Miniconda-latest-Linux-x86.sh
```

Execute commands below and follow the instructions to install miniconda:

```
chmod +x miniconda.sh
./miniconda.sh
```

You can now install some required and other useful dependencies for Poppy software with conda: `conda install numpy scipy ipython-notebook matplotlib`

You can now [install Poppy software](#).

25.2 Install the robotic simulator V-REP

[V-REP](#) is an efficient robotic simulator mainly open source (GNU GPL), which is distributed under a free licence for educational entities and have a commercial licence for other purposes.

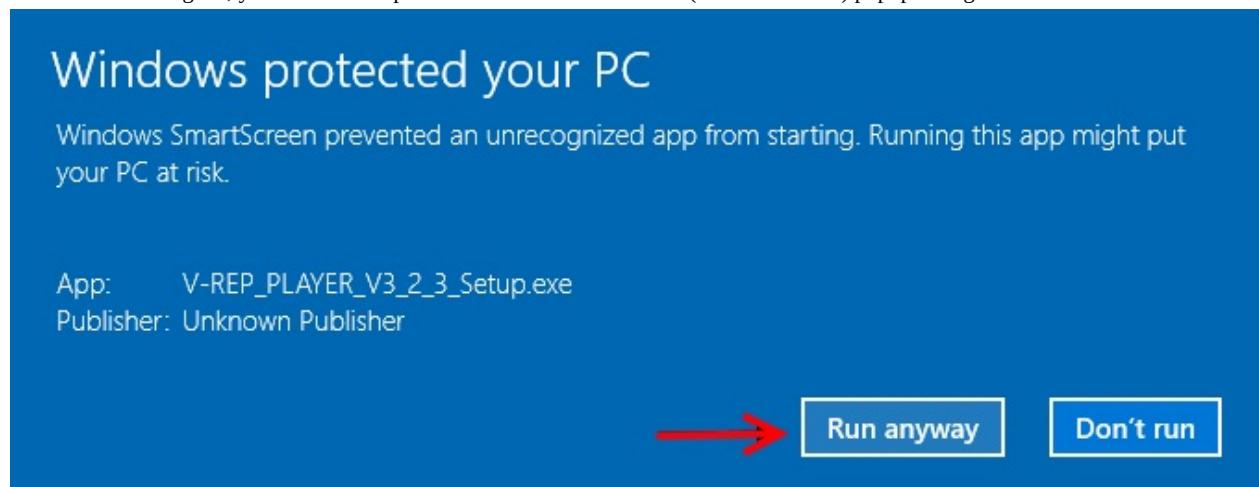
There is also an *PRO EVAL* version which limit the right to backup. As you don't need to backup the scene to use V-REP with Pypot (the Python library made for Poppy creatures), we suggest you to install this version to not worry about copyright infringement.

If you want to modify the V-REP scene for adding or customizing a Poppy creature, you will have to use the PRO or the EDU version (look at the [educational licence](#)).

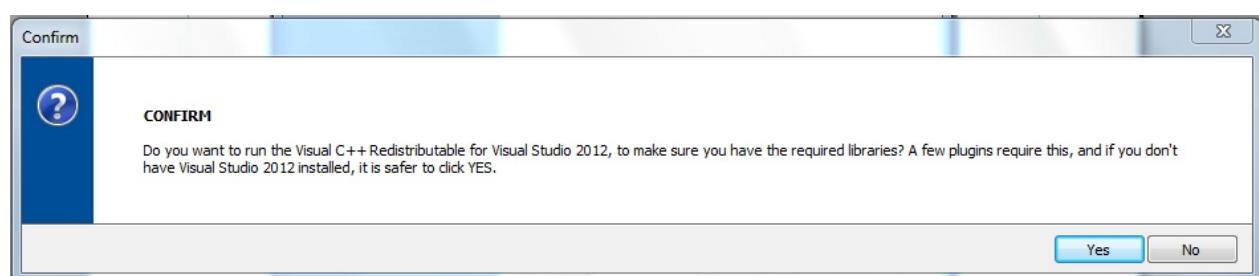
25.2.1 Install on Windows

[Download V-REP PRO EVAL or EDU](#) (if you are an educational entity).

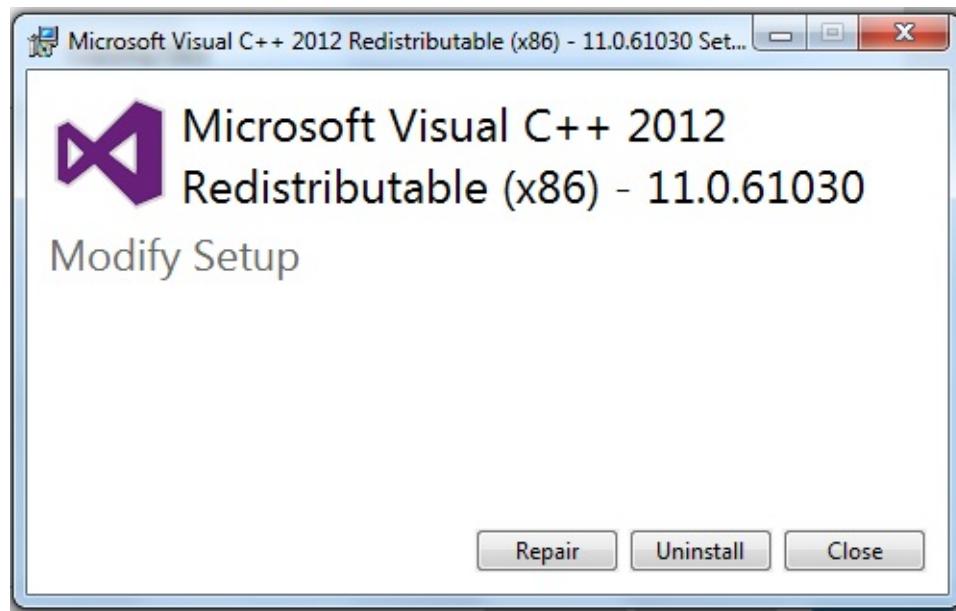
As V-REP is not signed, you will have to pass the Windows SmartScreen (on Windows 10) popup to begin the installation.



During the installation, make sure to install *Visual C++ Redistributable 2010* and *Visual C++ Redistributable 2012*.



Even if you already have *Visual C++ Redistributable 2010* or *Visual C++ Redistributable 2012*, it is advised to "repair" them (it is a re-installation process).



After the installation you can test if V-REP works well.

25.2.2 Test your installation

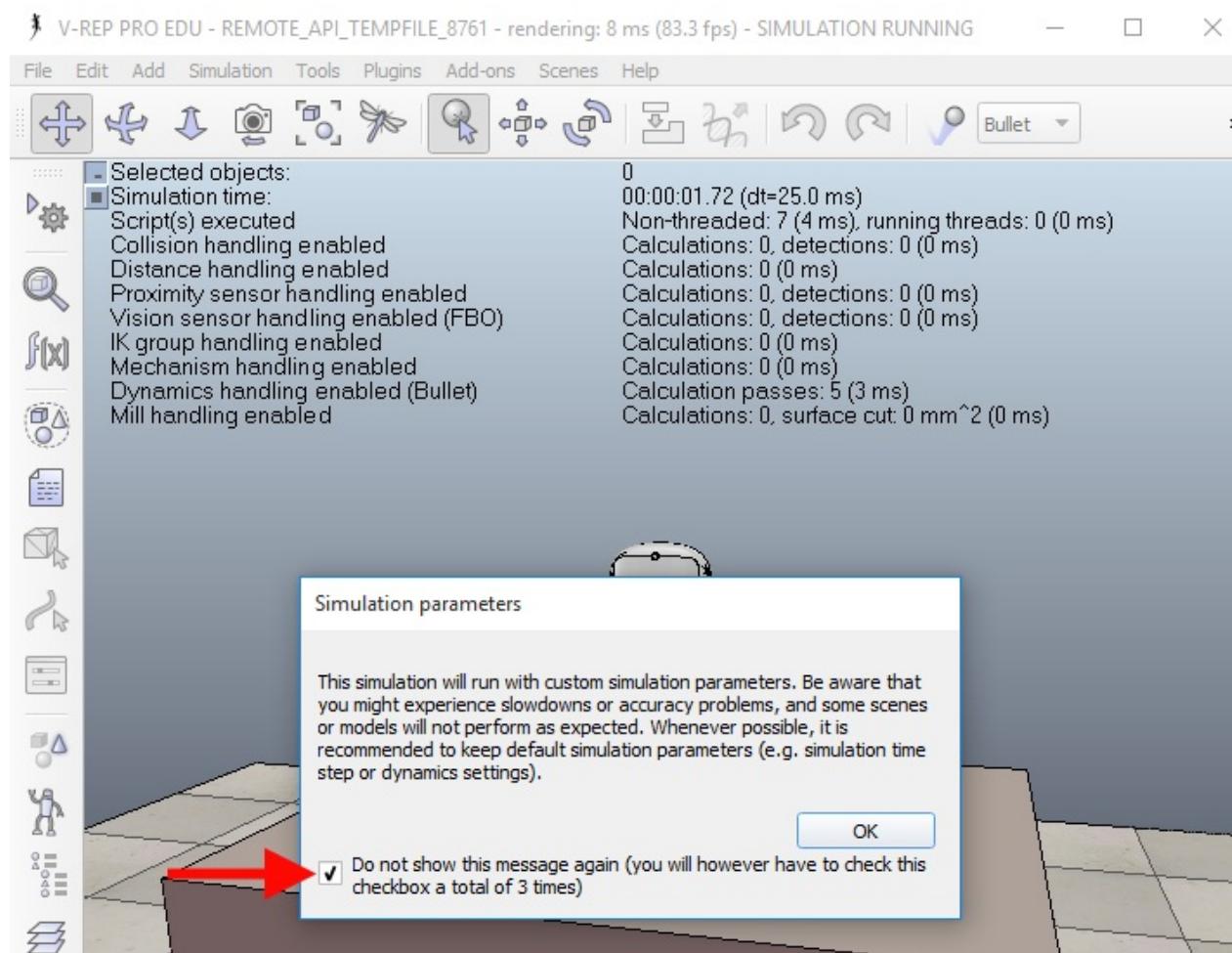
Open V-REP with a double click on the desktop icon.

Open the prompt of your Python Distribution (called *Anaconda Prompt* for Anaconda) or the *Command Prompt* of Windows, type and press Enter to execute the command below: `poppy-services --snap --vrep --no-browser poppy-torso`

After a few seconds, you will have an error like the picture below in your Command prompt.

```
C:\Users\poppy>poppy-services --snap --vrep --no-browser
No creature specified, use poppy-torso
Traceback (most recent call last):
  File "c:\users\poppy\miniconda2\lib\runpy.py", line 162, in _run_module_as_main
    "__main__", fname, loader, pkg_name)
  File "c:\users\poppy\miniconda2\lib\runpy.py", line 72, in _run_code
    exec code in run_globals
  File "C:\Users\poppy\Miniconda2\Scripts\poppy-services.exe\__main__.py", line 9, in <module>
  File "c:\users\poppy\miniconda2\lib\site-packages\poppy\creatures\services_launcher.py", line 75, in main
    poppy = installed_poppy_creatures[args.creature](**poppy_args)
  File "c:\users\poppy\miniconda2\lib\site-packages\poppy\creatures\abstractcreature.py", line 86, in __new__
    poppy_creature = from_vrep(config, host, port, scene)
  File "c:\users\poppy\miniconda2\lib\site-packages\pypot\vrep\__init__.py", line 75, in from_vrep
    vrep_io = VrepIO(vrep_host, vrep_port)
  File "c:\users\poppy\miniconda2\lib\site-packages\pypot\vrep\io.py", line 69, in __init__
    self.open_io()
  File "c:\users\poppy\miniconda2\lib\site-packages\pypot\vrep\io.py", line 80, in open_io
    msg.format(self.vrep_host, self.vrep_port))
pypot.vrep.io.VrepConnectionError: Could not connect to V-REP server on localhost:19997. This could also means that you still have a previously opened connection running! (try pypot.vrep.close_all_connections())
C:\Users\poppy>
```

If you switch to the V-REP window, a popup appeared to inform you that the simulation use custom parameters. This popup block the communication to the Python API of V-REP. **You have to check the check-box "Do not show this message again" and press "Ok".**



Switch the the command prompt window. You'll have to type the last command (poppy-services --snap --vrep --no-browser poppy-torso) and click to the vrep popup (with the checkbox checked) *three times* to make it works well!

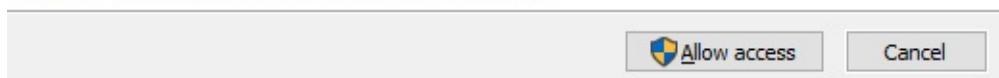
Note: to avoid retyping the same command again and again, you can simply press the up arrow key to call the last typed line.

Now type the last command without the "--no-browser" part. `poppy-services --snap --vrep poppy-torso`

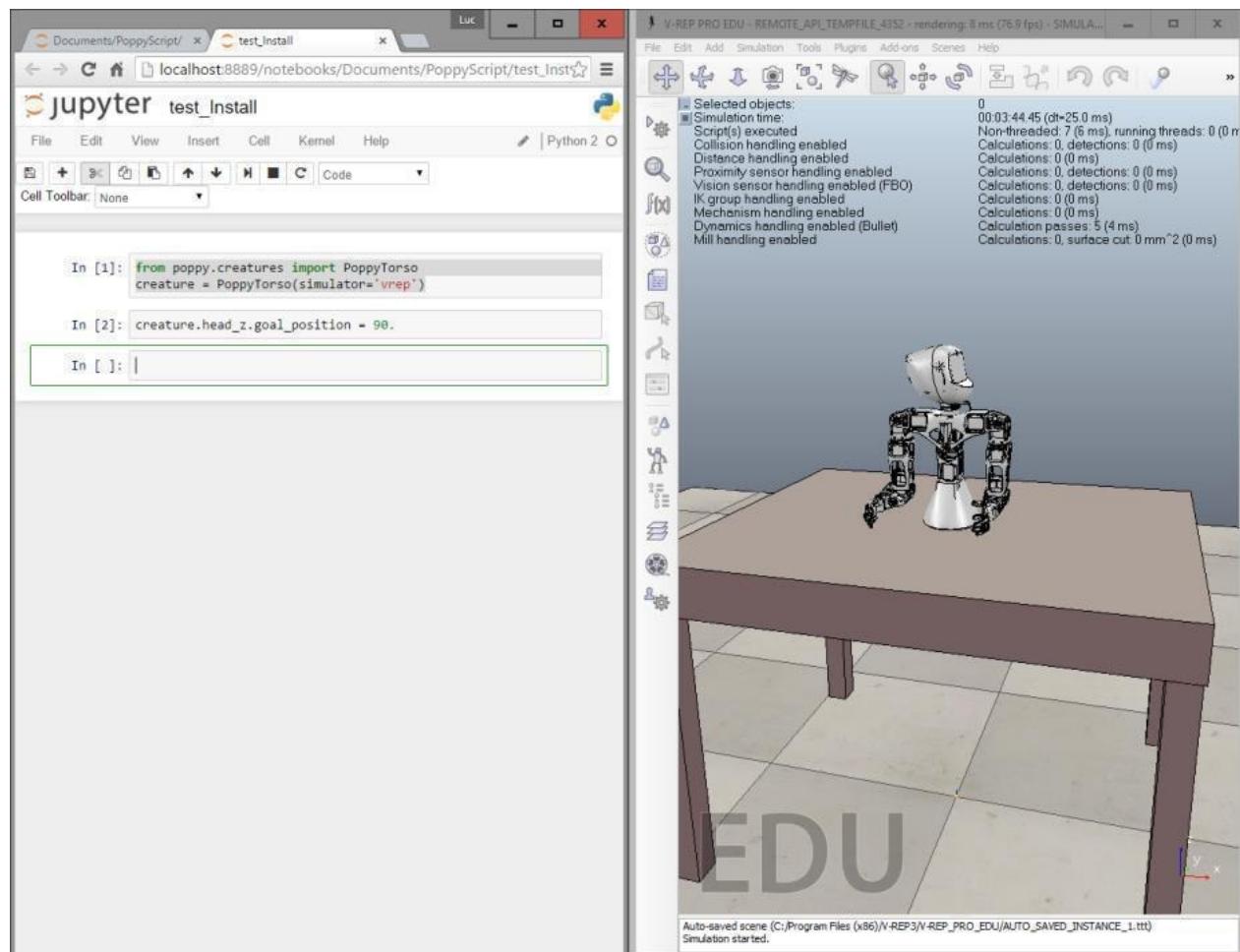
If you see a firewall popup like the picture below, be sure to check the "private network" checkbox.



What are the risks of allowing an app through a firewall?



If everything works, a new tab have been opened on your default web-browser. You can program you robot in Snap! or in Python.



25.3 Install drivers

Note: this chapter is only for people who want to control a tangible robot without an embedded board (Raspberry Pi or Odroid). It is a special case for advanced users

If you intend to control tangible robots from your computer **without** a Raspberry Pi or a Odroid, and you use a computer with Windows (vs GNU/Linux or MAC OSX), you may need to install manually drivers for the USB2AX or the USB2Dynamixel.

25.3.1 If you use a [USB2AX](#)

If the USB2AX is not recognized out of the box (its LED stay red after having been plugged) on your computer, you probably need to install manually its drivers.

The installation process and the files to download can be found on the [USB2AX documentation](#). You don't need drivers for GNU/Linux or MAC OSX, but note that it doesn't work very well with MAC OSX.

25.3.2 If you use a [USB2Dynamixel](#)

You need to install FTDI drivers on your computer. You have to low the "Latency Timer Value" from 16ms to 1ms (minimum allowed value) as explained in the [FTDI documentation](#) to avoid pypot timeouts.

26Switching from a simulated to a real Poppy robot

A key feature of the Poppy project is to let you, as seamlessly as possible, switch from a simulated robot (e.g. using V-REP) to a real one. It is particularly useful when:

- Developing an experiment where you can setup everything using the simulation, then run it on the real robot.
- Working in a classroom context where students can work on their own computer via the simulation and share a robot for real world tests

While it has been designed to cut the effort needed to switch from one to the other, there are still a few steps to understand. One of the major difference is when you are working in simulation everything runs on your own computer while when you are using a real robot, the software (e.g. the Python Notebooks) actually runs in the robot.

26.1 Using Snap!

TODO

27.1 Using Jupyter Python Notebooks

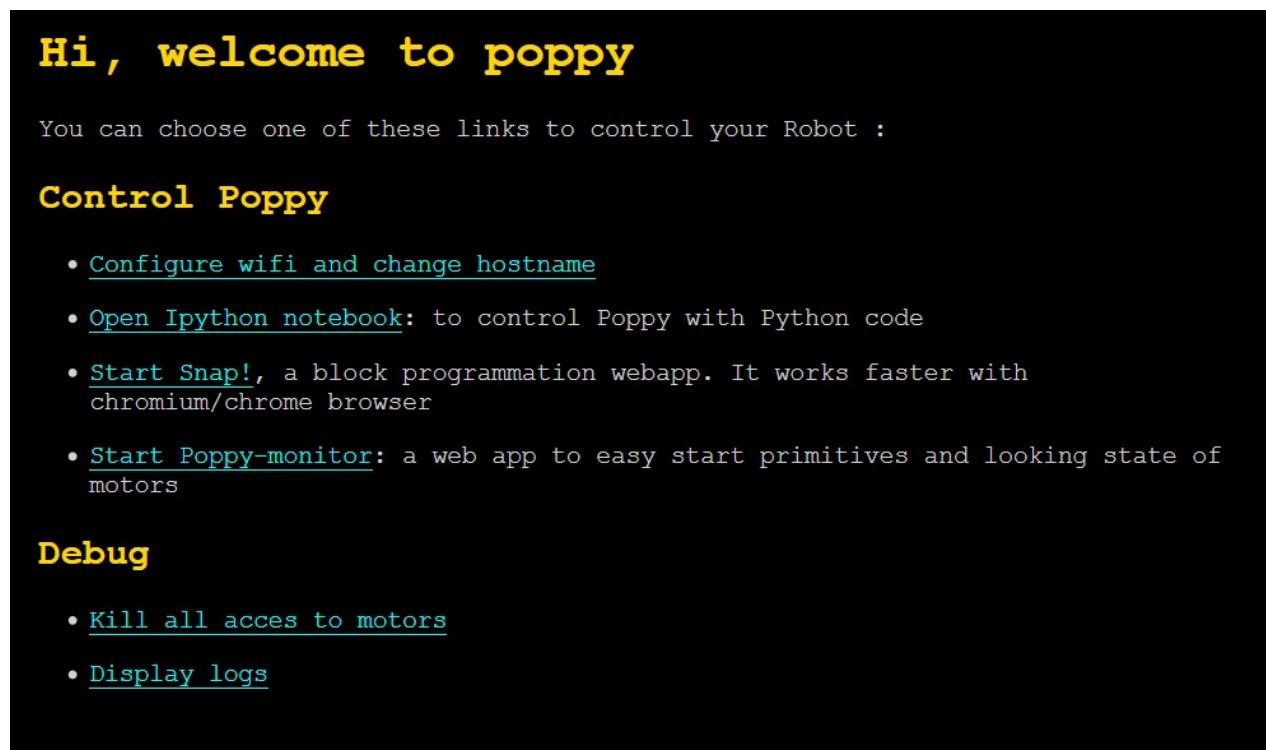
One of the advantages of working with Jupyter Notebooks is the possibility to use it as a client/server approach. Each Poppy robot hosts a Jupyter server accessible via the web interface (see [section quickstart](#) for details).

When working in simulation, everything is run and stored locally on your computer. When working with a real robot you can program it from a web browser on your own machine but your notebooks are actually stored and run in the robot.

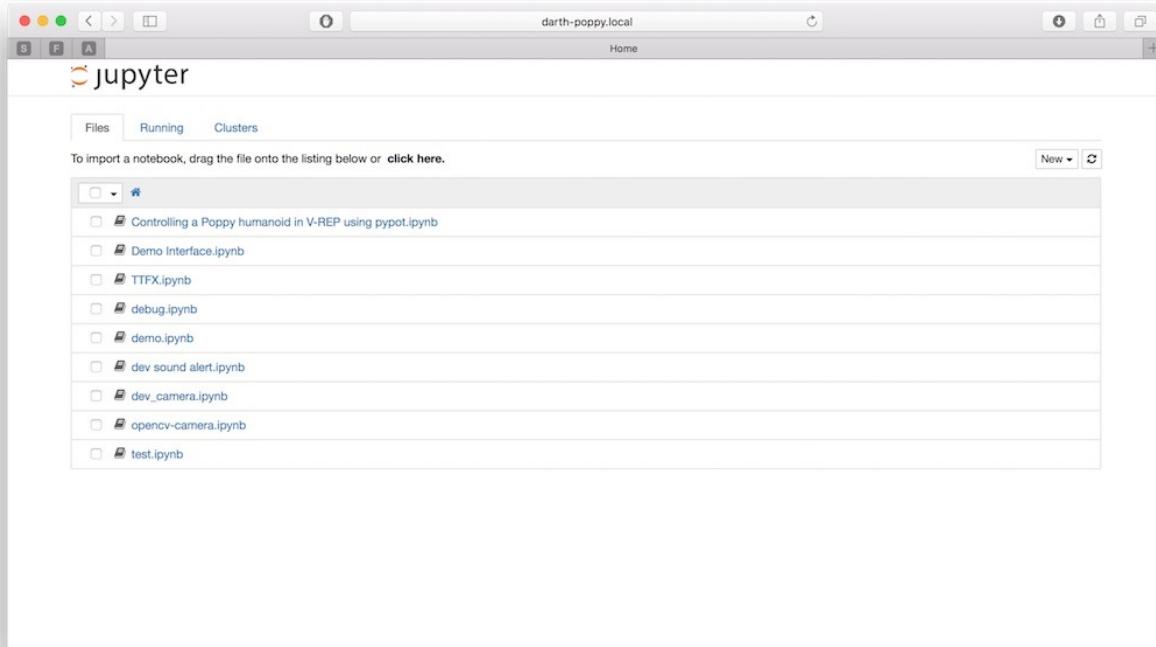
Thus to switch from simulation to a real robot, you need to switch from your local instance of Jupyter to the remote one hosted by the robot. The steps are described below.

27.0.1 Connect to the Jupyter on the robot

Once connected to the robot web interface (we will assume here its hostname is *poppy*, just replace it by the new hostname if you changed it), you should see a **open Ipython notebook** link.



When clicked it will start Jupyter on the robot and redirect you to the Jupyter webserver. You should then see the root of the notebook folder hosted on the robot:

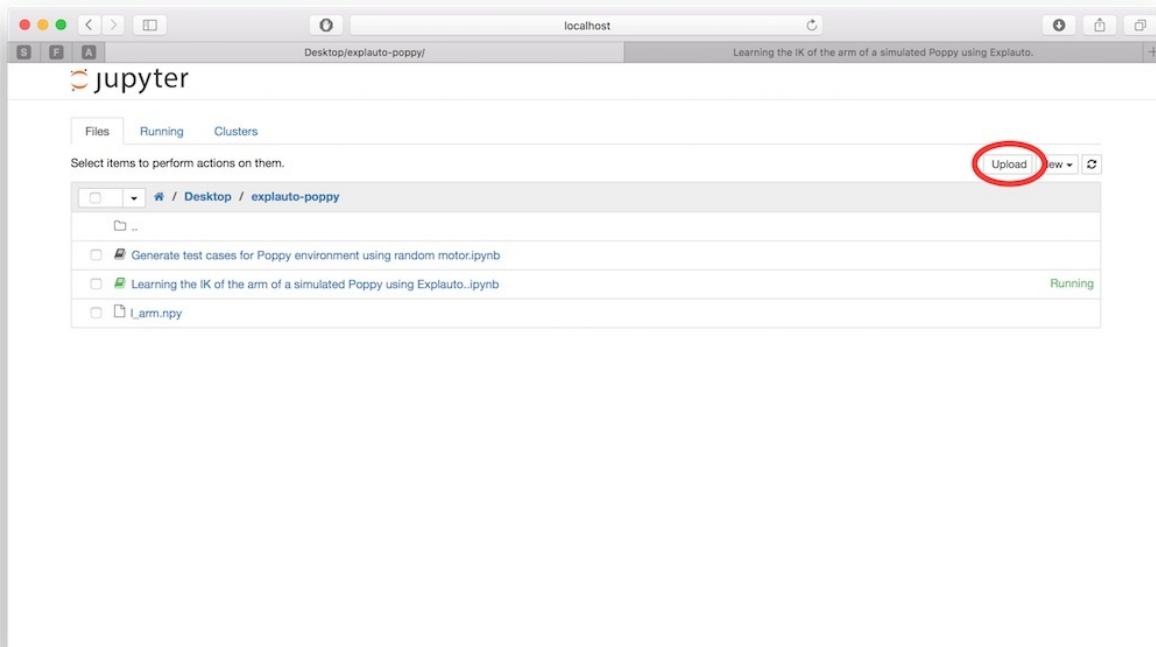


This is where you can put your own notebooks. Of course, you can create folder, organize them as you want, etc...

Note: If you need finer access or more advanced configuration (such as permission for instance), you have to log directly to the robot using ssh. See [advanced section](#) for details.

27.0.2 Upload a notebook

Once connected to the Jupyter server hosted by the robot, you can directly use the Jupyter interface for uploading new notebooks.



The circled button let you *upload* your local notebook, so stored on your own machine, to the robot. They can then be directly run on the robot.

Be aware that at the moment, we do not deal with sessions or permissions (as [JupyterHub](#) does for instance), and thus anyone with access to the robot can use or delete all notebooks stored in the robot.

27.0.3Adapt your code

There are few places where you should actually modify your code so it works with a real robot. We try to minimize the effort needed as much as possible, yet some steps are still required.

27.0.3.1Instantiation

When creating the robot, you actually need to specify if you are willing to work with a real or a simulated robot. This is simply done via a parameter. For instance:

When working with V-REP:

```
from poppy.creatures import PoppyHumanoid

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

Will become for a real robot:

```
from poppy.creatures import PoppyHumanoid

poppy = PoppyHumanoid()
```

Of course, this works for all existing Poppy creatures: Humanoid, Torso and ErgoJr.

This is most of the changes that you should do.

27.0.3.2Specific APIs

Some part of the API are platform specific. For instance, when using V-REP you have access to *tracking* features that let you retrieve any object 3D position. Of course, such method do not have a real world equivalent and thus are not available when working with a real robot.

A good practice if you want to write code compatible for both cases is to use the *simulated* property. It is automatically set to the correct value depending on how your instantiate your robot. For instance,

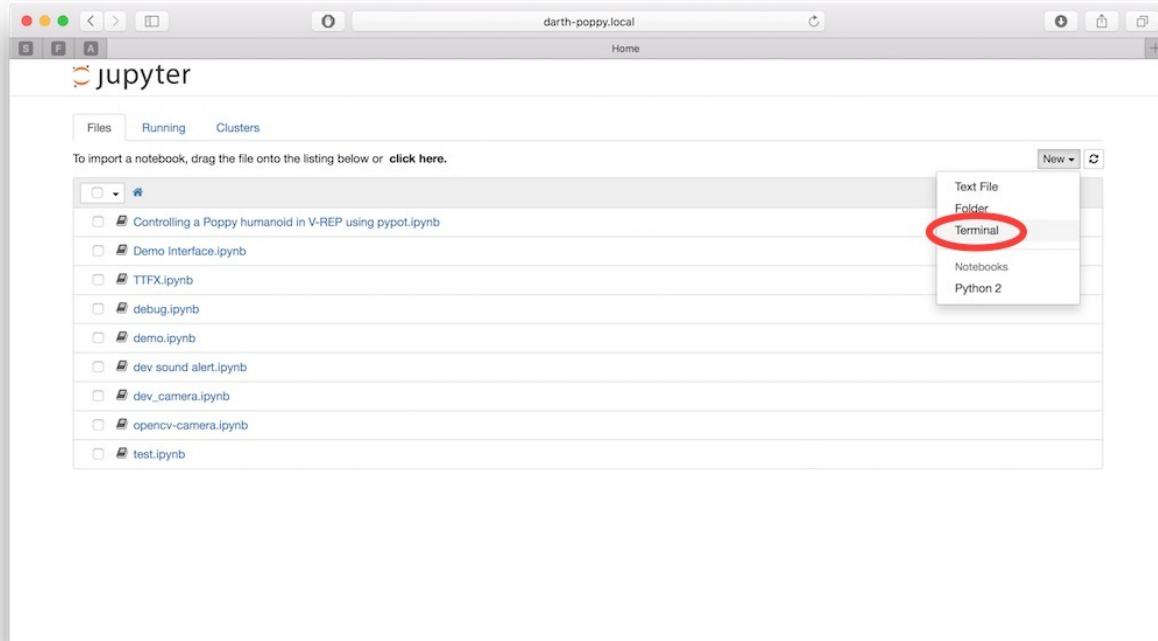
```
poppy = PoppyHumanoid(simulator='vrep')

def reset_position():
    if poppy.simulated:
        poppy.reset_simulation()
    else:
        print('Ask one internship student to put the robot back in its origin position.')
        time.sleep(10)
```

27.0.4Version and 3rd party libraries

The main drawback of this client/server way of working is that your locally installed software versions may differ from the one installed on the robot.

The Python installed on the robot is Python 2.7 and comes with most of the scientific main libraries (numpy, scipy, matplotlib, opencv). An exhaustive list of the installed Python packages will be available soon (TODO!). At the moment, the easier way to get it is to use a *terminal notebook* which can be directly run from the Jupyter interface.



```

poppy@darth-poppy:~$ pip show numpy
You are using pip version 6.1.1, however version 8.0.2 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
---
Metadata-Version: 1.1
Name: numpy
Version: 1.9.2
Summary: NumPy: array processing for numbers, strings, records, and objects.
Home-page: http://www.numpy.org
Author: NumPy Developers
Author-email: numpy-discussion@scipy.org
License: BSD
Location: /home/poppy/.pyenv/versions/2.7.9/lib/python2.7/site-packages
Requires:
poppy@darth-poppy:~$ 

```

Using the same technique, you can install 3rd party libraries directly on the robot. The `pip` and `conda` utility tools are installed and should be used when possible.

Note that the embedded board are based on armv7 and thus some libraries may be hard to compile. We are maintaining a list of conda recipes specifically built for this platform [here](#).

Contributions are more than welcomed!

29Software libraries

This section will provide software documentation of various libraries used in Poppy robots.

- [Pypot](#)
- [Poppy-creature](#)
- [Poppy Ergo Jr](#)
- [Poppy Humanoid](#)
- [Poppy Torso](#)

30Pypot library

Pypot documentation has still not been merged in the new documentation. You can find it at poppy-project.github.io/pypot/

31Poppy-creature library

31.1Introduction

Poppy-creature is a small library providing an abstract interface for robots (Poppy Humanoid, Poppy Torson, Poppy Ergo Jr...). It links high level controls and pypot and Pypot, the generic, lower level library.

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

The arguments you can provide are:

- `base_path` default: None Path where the creature sources are. The librarie looks in the default PATH if not set.
- `config` default: None Path to the configuration file with respect to the base-path
- `simulator` default: None Possible values : 'vrep' or 'poppy-simu'. 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
- `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
- `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

The sources are available on [GitHub](#).

31.2Poppy services

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-ergo-jr --snap --no-browser
```

This will launch the SnapRemoteServer for a real Poppy Ergo Jr robot.

The `--no-browser` option avoid the automatic redirection to the *Snap!* webpage. You can remove it if you use a computer with a GUI (e.g your laptop instead of the robot embedded board).

Another example:

```
poppy-services poppy-ergo-jr --poppy-simu
```

It will open a *Snap!* windows for a simulated poppy-ergo-jr.

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
- `--poppy-simu` : creates the specified creature for using with web simulator and also launches the HTTP server needed by poppy-simu. Poppy-simu is only available for poppy-ergo-jr for now.
- `--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)

31.3 Create your own Poppy creature

While developing a new Poppy creature, it is first easier to simply define it in a configuration file or dictionary and instantiate a `pypot.robot.robot.Robot` from Pypot directly.

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

Example from Poppy Humanoid:

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

        for m in robot.motors:
            m.goto_behavior = 'minjerk'

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

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

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

Package your code it properly using `setuptools`.

For a better integration with the Poppy installer scripts, please have in the root of your repo a folder named `software` containing:

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

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

31.4 Installing

poppy-creature library is a dependency of any Poppy robots libraries, so you don't have to install it by hand in a normal case.

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
```

32Poppy-ergo-jr library

This section need to be completed, contribution are welcome !

33Poppy-humanoid library

This section need to be completed, contribution are welcome !

34Poppy-torso library

This section need to be completed, contribution are welcome !

35 Appendix

35.1Getting involved

36.1 Raspoppo: manual setup of a Raspberry-Pi

1Glossary

0.1Compliant (for a motor)

A motor is said [compliant](https://en.wikipedia.org/wiki/Compliant_mechanism) if it opposes no resistance to a force. The opposite state of compliant is *stiff*.