COOPERA - User Manual

1 Introduction

COOPERA (COst-effective Open-source Platform for Empirical Robotics and Arficial Intelligence) is a robotic platform based on the *Bioloid* robot and it took inspiration from the *iCub* project. *Yarp*, a middleware software developed in the context of the iCub project, was used as a middleware layer to implement a modular and efficient interface between the user and the robot. With a modest economic investment (the overall cost of the platform is less than 1500$) it is possible to own a small and robust humanoid with 19 degrees of freedom with a camera, a 32-bit processing unit running linux-based kernel, and a YARP-based interface. COOPERA is able to demonstrate experiments involving artificial intelligence and machine learning algorithms applied to humanoid robots. In spite of its intrinsical limitations, COOPERA is an useful research tool. In particular, we have used this platform in order to program a robot that is able to learn directly on the real robot (in-vivo) how to stand up using a Reinforcement Learning approach instead of using simulators, as usually done in research.

In order to fully understand the content of this wiki it is required to have a basic knowledge of Yarp.

Readers that just want to use the platform to run the in-vivo RL experiments you just need to read Section 2 and 3. However if you wish to understand the platform details and desire to work on it by adding new modules and drivers or by enhancing the existing ones, read all the contents of this brief wiki.

2 Getting started

2.1 Software required

- **Yarp**: to enable the wireless communication between the client and the robot
- **SSH**: to enable remote communication with the robot board
- **JsonCpp**: for the configuration files (if BehaviorModule, SarsaPolicy, SarsaLearner, or MotorsController are used)
- **Shark**: for the linear classification performed during pose estimation (if BehaviorModule, SarsaPolicy, SarsaLearner, or MotorsController are used)

- **Boost**: for the Graph library (if BehaviorModule, SarsaPolicy, SarsaLearner, or MotorsController are used)
- **GNU/Linux**: Because of the two tools below and a few bash script used (if BehaviorModule, SarsaPolicy, SarsaLearner, or MotorsController are used)
- **speaker-test**: Linux tool to play the beep signal after a fault is detected (if BehaviorModule is used)
- **mpg123**: Linux tool to play a mp3 audio file when the battery is critical (if BehaviorModule is used)

2.2 Start up the robot interface

The steps that follow are necessary to boot the robot and start up the interface needed to operate with the robot through the Yarp modules:

- On a terminal, run a Yarp server with namespace *bioloid* and the following configuration: *192.168.1.100 10000 yarp*
- Connect the battery and switch on the robot controller
- Wait a few minutes for the Gumstix board to boot
- Connect remotely through SSH: *ssh root@192.168.1.80*
- Change the working directory: *cd BioloidYarp/build*
- Run the interface: *./BioloidYarp ../config.ini*. You should see a lot of Yarp messages being displayed. In this phase the drivers are initialized.
- If everything works you should see the message: *Everything is initialized!!*
- If the program is stuck with the message *0 devices found* means that the USB2Dynamixel (inside the head) is not properly plugged
- Other messages that can be useful to debug a problem in this phase are the ones regarding the Phidget. If everything worked you should see two messages: the first one (*Waiting for spatial to be attached....*) meaning that it is trying to detect a Phidget device, while the second one (*Spatial 129422 attached!*) meaning that it has been detected. If the second one is not displayed, is likely that the device is not properly plugged.
- If the program is stuck while displaying Yarp messages, it is likely that the robot is not properly working or there is a fault in a motor. Try switching the controller off and on again and restart the interface. If this does not work see Section 4.2.

3 BehaviorModule: in-vivo RL

The module that runs the SARSA algorithm to learn a task *in vivo* is called *BehaviorModule*. In order to run the module follow the steps:

- Change working directory to the build subdir: *cd BehaviorModule/build/*
- Run the program: *./BehaviorModule –from ../resources/conf/behaviorModule.ini*. Several Yarp messages will be displayed.
- If everything worked you should see the last message: *yarp: Port /BehaviorModule active at tcp://192.168.1.100:10028*, otherwise it is likely that the robot is not properly working or there is a fault in a motor. Try switching the controller off and on again and restart the interface. If this does not work see Section 4.2.
- On a new terminal type: *yarp rpc /BehaviorModule*
- To run a policy for the task specified in the configuration file (discussed in Section 3.5), type the command *learn*
- To exploit the policy learned, type the command *exploit*

3.1 Stop, resume and reset the learning

For long lasting learning, such as the standing up learning, it will be required to stop the learning and resuming it (e.g., when the battery level is critical). This is possible in any moment (the learner records its advancement at the end of each episode). However, it could be the case that you don't want to resume the learning at all because you have changed some parameters and now you want to start the learning over from the beginning. In this case, you will have to type the command *reset* before restarting the learning with the command *learn*.

3.2 Alarms

Two are the alarms sent out by BehaviorModule:

1. When the battery level is critical (a mp3 audio file devised to draw immediate attention, generated by the tool *mpg123*).
2. When a hardware fault is detected (a beep, generated by the tool *speaker-test*)

In the first case it is required to forcefully shut down the interface, *halt* the system and unplug the battery. However, when you first listen the audio signal is likely that the battery is not critical yet but the voltage went down temporarily. Only when you hear multiple audio signal consecutively it means the battery level is critical. As to the battery life, it is about 50 minutes when performing motors movements for the whole time whereas when the motors are not used at all it last up to 70 minutes. The recharge process (using the HYPERION charger) takes approximately one hour. It is also possible to power the robot plugging it to the AC.

In the second case, that is when a fault is detect, the procedures to follow depend on the type of fault and are explained in Section 4.2. The module automatically record the faults occurred in files in the *build* directory: *overload.txt*, *overheating.txt*, and *angleerror.txt*. Since the firmware fails to deliver a message when the fault cause are the encoders, when the cause is unknown it is considered to be an angle error, even though this assumption might be wrong.

3.3 Actions specification

The actions specification follows a hierarchical structure. The files that must specified in the configuration file discussed above are *macro* files. A macro is a combination of simple *behaviors*, which in turn define, within a specific robot part, which joints to move. The behaviors are parametrized accordingly to the *Discretizations* defined in the configuration file. The macros are stored in the directory *resources/data/macros*, whereas the behaviors are stored in the directory *resources/data/behaviors*. The several behaviors that form a macro are executed simultaneously; analogously the several joints that are concerned by a single behavior are executed simultaneously. The structure of a macro file is straightforward: it is a list of behavior file names. A single behaviors specification includes a line for each joint used; each line specifies the concerned robot part, the joint and the movement direction.

3.4 State specification

The state specification, for this module, includes the data from the encoders of the most relevant motors as well as the measured tilts from the Phidget sensor we equipped the robot with. The conjunction of these data provide a reliable estimate on the robot state. Therefore, the state vector used in the module is:

Motor 0	Motor 1	Motor 3	Motor 4	Motor 6	Motor 7	Motor 8	Motor 9
Motor 10	Motor 12	Motor 13	Motor 14	Motor 15	Motor 16	TiltX	TiltY

However, you do not have to pay attention to which motors have to be removed when writing the initial states and target state in the configuration file because they are automatically filtered out.

3.5 Configuration file

Ini file

A INI file is used to specify parameters concerning the Yarp module developed. It follow a quick explanation of such parameters.

- **name**: The module name.

- **robot**: The robot name.
- **defaultvelocity**: The default velocity.
- **parts**: List containing the robot parts used.
- **csv**: Path for the file used as training set for pose estimation.
- **njoints**: List containing the number of joints for each robot part.
- **batterylimit**: Lowest voltage allowed for the battery.
- **batteryindex**: Sensor's index to be used to get the battery value.
- **batteryalarm**: mp3 file to be used as an alarm or empty for a beep.
- **acclimit**: Threshold used for fall detection.
- **accindex**: Sensor's index to be used to get the acceleration values.
- **tiltindex**: Sensor's index to be used to get the tilt values.
- **motionwait**: Time to wait for completion of each motion during descent.
- **macrowait**: Time to wait after an entire macro.
- **poseestimate**: Whether to use pose estimation or not.
- **pose**: Pose to use for initial position if not estimated.

Two parameter are worth noting: *pose* and *poseestimate* are, by default, respectively *facedown* and *false*, meaning that pose estimation is disabled and it is used the face down position. In order to switch to pose estimation change the *pose* parameter. By doing so, at the end of each episode, the reset feature will take the robot in a random initial pose and the a policy will be generated for all of them. Another parameter that you might want to change is *macrowait*. By default it is set to 5 seconds, however if you want to introduce more complex actions it could be a good idea to increase this parameter as well, otherwise you might experience overlapped actions.

Json file

A JSON file is used to specify parameters concerning the RL algorithm. In this section I am going to explaining some of the parameters found in the JSON configuration file, that are the ones that I have introduced. For the parameters not specified here, ask Shashank. Please note that you will need two JSON files: a *Config.json* file used during learning and a *base.json* file that is used for the *reset* mechanism. The two files must be equals before starting the learning for the reset command to do its work properly.

- **Domain/Actions**: it defines the available actions as a list of strings. Each string is a file where the action is specified. The directory it is scanning to find these actions is the *resources/data/macros/* subdirectory
- **Domain/Discretization**: it defines, for each joint plus the two acceleration feedback, the allowed values that they may achieve. Note that the size of these vectors times the size of the actions vector yields the number of total different actions available.
- **Domain/Init**: it defines the initial states (one for each of the automatically detected position). The values represent all the 18 encoders plus the two values from the accelerometer
- **Domain/Locked**: it is the list of the joint yet to be released (automatically updated itself after each releasing)
- **Domain/Target**: the specification of the target state. Same format used for Init
- **Experiment/EpisodesCompleted**: it is automatically updated for keeping track of how many episodes have already been completed (in order to resume the learning after an interruption)
- **Experiment/NSamples**: it automatically records how many random numbers have already been generated (for resuming learning after interruption)
- **Experiment/StatesCounter**: it automatically keeps track of how many states have already been generated
- **Experiment/SimulatedActions**: it automatically records how many actions have already been simulated
- **Learn/releaseJointAfter**: it specifies how often a new joint must be released
- **Experiment/NRealSamples**: how many times to run an action in a particular state before starting to simulate it

If the module is not able to learn an optimal policy for your selected task you should try to tune a few critical parameters, such as the ones concerning the exploration policies, the rewards, the simulated actions, the locked joints and so on. Also, if SARSA does not seem able to reach in any case the target state it might be that either the available actions and the allowed values for these actions are not enough and therefore define further actions or changing the *Discretization* parameter could help, or you might want to increase the *MaxLengthFactor* parameter that guides the maximum length of the episodes.

3.6 Faults management

In order to enable the robot to operate autonomously as long as possible it is crucial to consider and manage accordingly all the possible faults or exceptions that can arise. Whichever is the case, we are able to manage these exceptions by, at most, briefly assisting the robot. It would correspond to a mother that assists her child, which is not considered to cheat mother nature: nobody expects babies to be independent from birth. However, for most of the case, human assistance is not even required. A large part of the faults are due to overloading on the servomotors. By detecting high torques on them we are sometimes able to stop them before the fault occurs. In the cases when this is not possible we can nevertheless reset the torques and having the motors working again without any human intervention. Thanks to this prevention, detection and recovery strategy we were able to reduce drastically the rate of such faults.

4 General information about the platform

4.1 Other modules implemented

- ***Encoders***: can be used to get the encoders corresponding to a specific robot configuration or to set a new configuration by passing the angles that must be gained by the motors (for instance it was used to devise the BehaviorModule's reset procedures)
- ***Torques***: can be used to get the torques corresponding to a specific robot configuration
- ***Sensors***: can be used to get the sensors feedback corresponding to a specific robot configuration
- ***MotorsController***: the module that provide the initial demo: pose estimation(linear classification), turning (SARSA) and standing up (scripted)
- ***TrainTorques***: used to build the training test for the linear classification, used in MotorsController.
- ***SarsaLearner***: learning of a policy for turning the robot, used in MotorsController
- ***SarsaPolicy***: used to debug SarsaLearner
- ***Actions***: used to create (or test) actions, which are then used in MotorsController

4.2 Recover the robot from a fault

The Dynamixels the robot is equipped with can fault for several reasons:

- Too much load

- Missing feedback
- Heating
- Not enough voltage supply

Sometimes is enough to switch the controller off and on again to recover from a fault: it is the case when a torque error is triggered, that is when the load applied to a motor is too much. Actually, it is often possible to recover from this fault without any human intervention by resetting the torque of the affected motor and wait a few seconds before using the motor again (as done in BehaviorModule). If this procedure does not work, it is most likely that the position control feedback embedded in the Dynamixels had a fault. To fix these kind of problems it is required to manually change the motors position before switching the controller on. If the faulty feedback is instead a sensor it might be required to restart the robot main interface after being sure all wires are properly plugged. Particular care has to be used for the motor temperature: if the motor is overheated it is best to switch off the controller and wait until it reaches a reasonable temperature again. Clearly, also when there is not enough voltage supply to the motors, they will necessarily stop working.

4.3 Dynamixel Management

When it is required to manage the robot motors (e.g., for changing the id or other parameters) you will need to use the ROBOTIS suite *RoboPlus*, in particular the tool *Dynamixel Wizard* can be used for this purpose. In order to use this software you can either connect the entire robot to the Windows PC or you can isolate one motor and connect it singularly. In the first case it is enough to unplug the USB2Dynamixel from the hub located on the robot head and plug it on the PC. If you need instead to connect a single motor, you will have to remove the USB2Dynamixel from the robot head, connect directly the Dynamixel to the motor and then plug it on the PC. In addition, you will need to change the USB2Dynamixel switch to TTL (instead of RS232) and connect the motor to the controller as well. Additional information as to use these tools can be found on the ROBOTIS on-line help.

5 Replicate COOPERA

COOPERA is a Bioloid robot with the head removed and replaced by a box containing a Gumstix board, powered by a Linux distribution. If you want to replicate the platform you will need to build your own *head* and place it on the top of the robot. Moreover, in order to interact with the robot in the more comprehensive way, a custom firmware must be installed on the Bioloid controller.

5.1 The CM-510 custom firmware

The firmware installed in the Bioloid controller, which allows us to interact with sensors as well as motors, through serial connection is developed by Jose Cortes (*joscorare@gmail.com*). If you require to update or change the firmware refer to the *ROBOTIS* online help.

When using Jose's custom firmware you can have three mode:

- **Device Mode**: down button in the CM510 controller
- **Debug Mode**: left button in the CM510 controller
- **Toss Mode**: up button in the CM510 controller; it allows to interact with motors (id = 1 to 18) as well as sensors and CM510 itself (id = 200)

  For more information on this firmware contact the developer or see the website at:

  http://softwaresouls.com/softwaresouls/2011/11/22/cm-510-as-a-multiple-sensor-dynamixel-item/.

  You might need to sign up.

5.2 The robot embedded board

An embedded PC (Gumstix Overo) is mounted on the Bioloid (located in replacement to the original Bioloid head) in such a way that a Wi-fi connection can be used to communicate with it. The board is then connected to the Bioloid by means of the *USB2Dynamixel* device, which makes possible to interact with the robot, directly from the PC, through a serial connection.

The hardware used are the followings:

- Gumstix Overo Tide COM (Computer-On-Module)
- Gumstix Tobi expansion board
- SD card
- Wi-fi antennas
- USB2Dynamixel

The box used to replace the original robot head hosted all the hardware as well as a heat-sink for dissipation and a small USB fan, otherwise the COM would critically heat up in a few minutes. Furthermore, a USB hub turned out to be necessary in order to provide supplementary USB ports as well as to solve powering issues.

A Linux distribution that fitted our purposes, namely boot and load the drivers for the Wi-fi connection and for hosting the Yarp framework, is installed on the board. The Linux distribution used is available at:

cumulus.gumstix.org/images/angstrom/misc/caspax

In order to get a bootable memory for the board is enough to format a SD card with it by following the instructions available at:

gumstix.org/getting-started-guide.html

The distribution, called *Angstrom*, runs a 2.6.34 kernel. At the time of the development there were no better options at hand, however are now available *Ubuntu* and *Android* images.

6 Developer details

6.1 Drivers

Two Yarp drivers were developed for this platform:

- CM510Sensor
- DynamixelAX12BioloidFtdi

Both these drivers use a common channel to interact with the robot: FTDI. The communication goes through a device called USB2Dynamixel that allowed us to connect the Bioloid robot with the Linux board placed on the robot head. *CM510Sensor* deals with all the robot sensors (both the stock Bioloid sensors and the new installed sensors). It basically keep sending out these values through a Yarp port (*/bioloid/sensors*). The other driver implemented, *DynamixelAX12BioloidFtdi* deals with the robot motors and provide the basic features in order to control the Dynamixel servomotors the Bioloid robot is equipped with.

To modify or access these drivers:

- Connect remotely through SSH: *ssh root@192.168.1.80*
- The motors' driver is found at the path: *yarp-2.3.22/src/modules/DynamixelAX12BioloidFtdi*
- The sensors' driver is found at the path: *yarp-2.3.22/src/modules/CM510Sensors*
- Recall that if you edit these drivers, you need to recompile Yarp

6.2 Code documentation

The documentation for all the source code involved, generated with Doxygen, can be found in the home directory of the Gumstix's SD card (subdirectory *doc*).