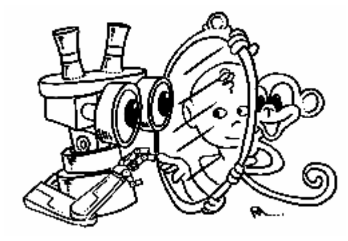


MIRROR

IST–2000-28159

Mirror Neurons based Object Recognition

Deliverable Item 2.5
Architecture of the learning artifact

Delivery Date: April 30, 2003

Classification: Public

Responsible Person: Prof. Giulio Sandini – DIST, University of Genova

Partners Contributed: ALL

Short Description:

This deliverable describes one of the outcomes of Mirror, i.e. the robotic implementation of the mirror neuron model. It includes definition of the hardware with particular emphasis to the robotic hand, which was realized as a part of the project. This document contains also a description of the low-level software layer, a behavior based componentized software structure that provides robot control functionalities and communication within a distributed architecture based on Pentium class processors and running a variety of operating systems. Further, this document includes the description of a learning model that to some extent has been repeatedly used within the architecture. Finally we present some of the latest experiments based on this model and point out at future integration with the mirror neuron model presented in deliverable D3.4.

Information Society

Project funded by the European Community under the "Information Society Technologies" Programme (1998-2002)

Content list

1. Introduction 2
2. Babybot – the bare robot hardware 3
3. Interface cards 6
4. OS independent software interface 7
5. Robot independent code 9
6. Robot specific interface 10
7. Learning architecture 10
8. Specific experiments 12
   8.1. Learning a visual model of the hand 13
   8.2. Learning gravity compensation 15
   8.3. First grasping strategies 17

1. Introduction

This deliverable describes one of the outcomes of Mirror, i.e. the robotic implementation of the mirror neuron model. For explanatory purpose the complete model can be divided in two parts covered by this document together with D3.4. The latter covers only the definition of the model of the mirror neurons proper. This document describes the robot architecture in its entirety. It includes definition of the hardware with particular emphasis to the robotic hand, which was realized as a part of the project. It contains also a description of the low-level software layer, a behavior based componentized software structure that provides robot control functionalities and communication within a distributed architecture based on Pentium class processors and running on a variety of operating systems. Further, this document includes the description of a learning model that to some extent has been repeatedly used within the general architecture. Finally, we illustrate some of the experiments and point out at future integration with the mirror neuron model detailed in D3.4.

Overall, the robot architecture can be seen as a layered system as shown in Figure 1. It is worth stressing here that layering involves only the "engineering" of the robotic artifact. It does not constrain what specific learning schema is employed or how the different developmental modules interact one with another. In this sense this is not in contrast with the main philosophical underpinning of the project: that is, certain cognitive skills have to be acquired through a prolonged developmental period. Although modularity (and layering in this case) is perhaps the only tool we have in face of complexity, we previously suggested that we need to carefully take into account how interrelationships between modules evolve as artificial adaptation unfolds if we are to properly characterize learning and development. It was important thus not to commit too early into the decision of which learning schema or control structure had to be used.

Bare robot hardware

Interface cards

OS independent software interface

Robot independent code

Robot specific interface

Learning architecture

Specific experiments

Figure 1 Organization of the hardware and software layers of the robotic artifact

The seven layers in Figure 1 span the conceptual space from the "bare metal" to the "experimental level". All what we have designed in between is sort of "supportive" structure. These middle layers might condition, to a certain extent, whether a particular type of experimentation could be actually carried out, and consequently they represented a delicate design decision.

The first layer contains the robot hardware. It includes a simplified description of the mechanics and the limitations of the current implementation. The robot's sensors and actuators are interfaced to a set of computers through interface cards. Layer number 2 is made of these cards.

Layer 3 is the first software interface with the hardware. It is written with portability in mind since many different cards might be connected to the same setup [or, on the contrary, the same card might be used in another setup], the controlling computers could be possibly running different operating systems, and different subparts might be running on different machines connected through an IP-based network. Using a publicly available portable library (ACE), we designed a communication system and general purpose computation environment. Our guiding criterion was “try not to be monolithic”. In this sense our system does not force the use of any particular library or set of libraries and in fact it was easily interfaced to Intel IPL (for efficient image processing), and Matlab (because of the many available toolboxes). The core system ran seamlessly on Windows (NT series), QNX (a real-time OS), and Linux.

In general, code directly controlling the robot tends to have a nasty organization and often contain details of the specific locale even when the controlling cards are the same. We wanted to decouple the “robot” part of our software from the more general purpose OS interface. Consequently, layer 4 is meant as an interface between the robot-independent code built on a generic “device driver” definition [device driver here is a user-level software component]. The goal of the latter is of shielding the nasty details from the higher level where unspecific code could be produced. This allows interfacing to any different hardware at the price of rewriting a relatively small virtual device driver module. Layer 5 allows customization of the robot control code to the specific locale. This is required in order to “tune” the control code to the details of the actual robot. It allows reusing the same code on two not-too-dissimilar setups. It consists mainly on specific software and configuration files.

Layer 6 contains a general description of the organization of the learning modules of the system. These components are simply a tentative formalization of the implementation of the sensorimotor coordination behaviors of the robot. At this stage they are not necessarily innovative in any respect and in fact we pretty much relied on standard function approximation methods and traditional neural network algorithms (e.g. backpropagation).

Finally, layer 7 represents the level where specific experiments are conducted. It is worth noting that none of this organizational structure is strictly imposed. From the outside the architecture consists of a set of libraries and executable modules. The remainder of this document is divided into sections describing each layer to a reasonable level of detail.

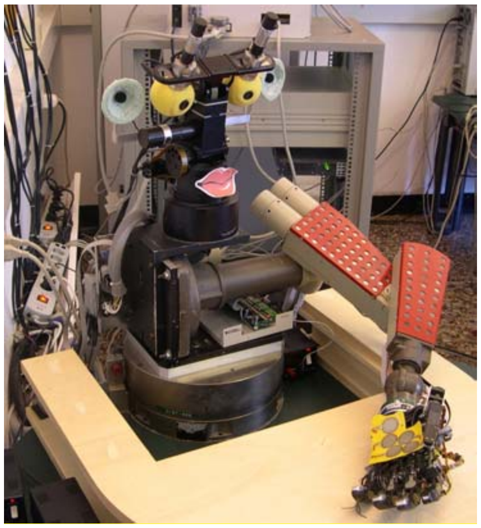


Figure 2 The robotic setup, the Babybot.

2. Babybot – the bare robot hardware

Babybot is an upper torso humanoid robot with a head, a manipulator arm and a hand. The head has five degrees of freedom (DoF). Three of them are associated with the two cameras to achieve independent panning and coupled tilting. The two remaining DoF allow panning and tilting at the level of the neck respectively. The manipulator is a 6 DoF Unimation PUMA 260, mounted with the shoulder horizontal to better resemble a human like arm (see Figure 2). In practice since it lacks a degree of freedom in the shoulder it often requires awkward configurations to reach a point in space.

Attached to the arm end point is a 5 fingered robot hand. Each finger has 3 phalanges; the thumb can also rotate toward the palm. Overall the number of degrees of freedom is hence 16. Since for reasons of size and space it is practically impossible to actuate the 16 joints independently, only six motors were mounted in the palm. Two motors control the rotation and the flexion of the thumb. The first and the second phalanx of the index finger can be controlled independently. Medium, ring and little finger are linked mechanically thus to form a single virtual finger controlled by the two remaining motors. No motors are connected to the fingertips; they are mechanically coupled to the preceding phalanges in order to bend in a natural way. This is explained in Figure 3.

The mechanical coupling between gears and links is realized with springs. This has the following advantages:

- The mechanical coupling between medium, ring, and small finger is not rigid. The action of the external environment (the object the hand is grasping) can result in different hand postures (see Figure 5).
- Low impedance, intrinsic elasticity. Same motor position results in different hand postures depending on the object being grasped.
- Force control: by measuring the spring displacement it is possible to gauge the force exerted by each joint.

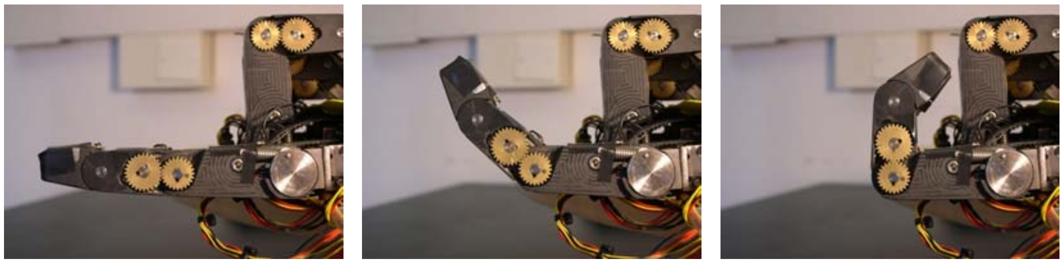


Figure 3 Mechanical coupling between the second and the third phalanges. The second phalanx of the index finger is directly actuated by a motor. Two gears transmit the motion to the third phalange. The movement is respectively of 90 and 45 degrees.

The robot's sensory systems include vision, audition, touch, proprioception, and inertial sensing. Proprioceptive feedback is achieved by means of the motor optical encoders. Two cameras rotating with the eyes and two microphones attached to the head respectively provide visual and auditory feedback. During the acquisition, images are sampled in non-uniformly to mimic the distribution of receptors of the human retina. More pixels are acquired in the central part of the image (fovea) and less in the periphery (mathematically the distribution is approximated by a log-polar function).

The head mounts a three axis gyroscope that provides the robot with an artificial equivalent of the human vestibular system (in Figure 4). This sensor measures inertial information consisting of angular velocity along three orthogonal axes. It can be used for stabilizing the visual world efficiently and in coordinating the movement of the head with that of the eyes.

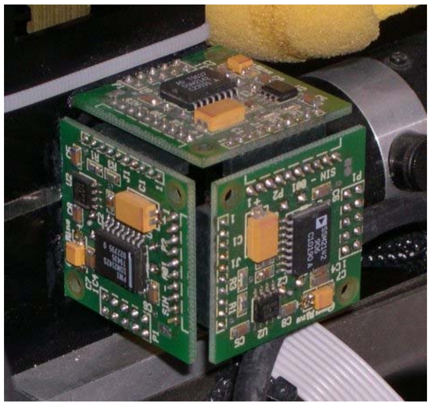


Figure 4 The inertial sensor of the Babybot developed at LIRA-Lab. It consists of three mono-axial sensors arranged along three orthogonal axes.

For the hand, Hall-effect encoders at each joint measure the strain of the hand's joint coupling spring. This information jointly with that provided by the motor optical encoders it is in theory possible to estimate the posture of the hand and the tension at each joint. In addition, force sensing resistors (FSRs) are mounted on the hand to give the robot tactile feedback. These commercially available sensors exhibit a change in conductance in response to a change of pressure. Although not suitable for precise measurements, their response can be used to detect contact and measure to some extent the force exerted to the object surface. Five sensors have been placed in the palm and three in each finger [apart from the little finger] (see Figure 4).

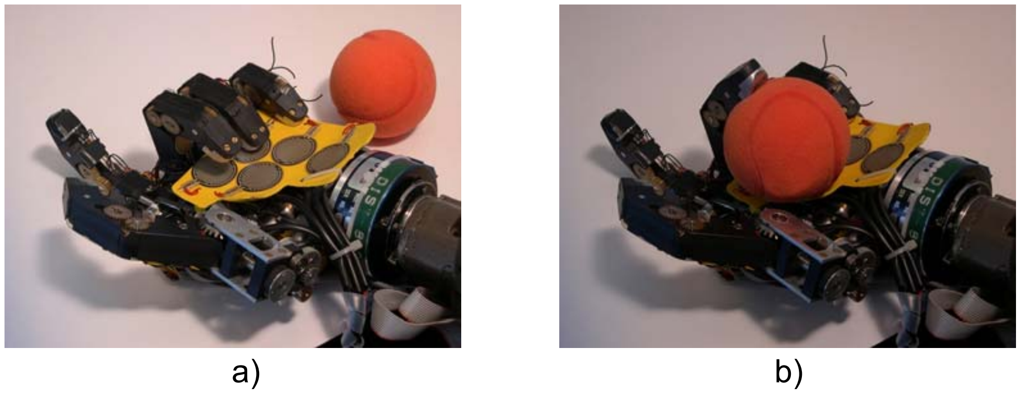


Figure 5 Elastic coupling. a) and b) show two different postures of the hand. Note however that in both cases the position of the motor shafts is the same. In b) the intrinsic compliance of the medium finger allow the hand to adapt to the shape of the object.

Further proprioceptive information is provided to the robot by a strain gauge torque/force sensor mounted at the link between the hand and the manipulator's wrist. This device is a standard JR3 sensor designed specifically for the PUMA flange. It can measure forces and torques along three orthogonal axes (see).

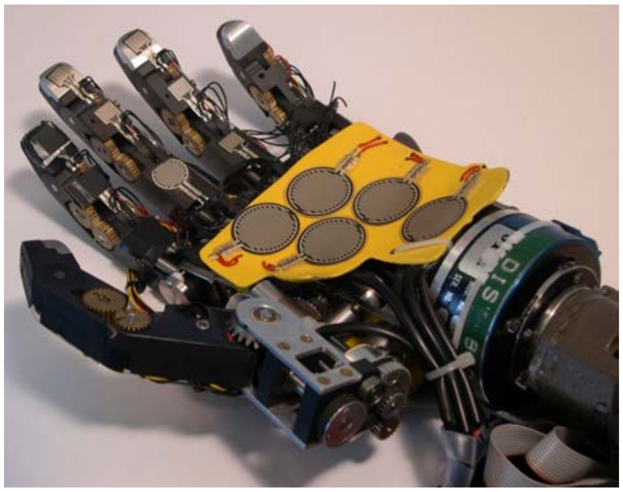


Figure 6 Tactile sensors. 17 Sensors have been placed: five in the palm, three on each finger apart the little finger. In this picture the sensors in the thumb are hidden. The short blue cylinder that links the PUMA wrist to the hand is the J3R force sensor.

3. Interface cards

The robot sensing includes some digitizing interfaces and special signal conversion and conditioning modules. The link between the hardware and the robot is provided through standard PCI/ISA cards, serial ports, etc.

Motor control also requires special hardware to generate the appropriate signal piloting the motors. At this level the Babybot follows a very traditional engineering approach. The robot is actuated by DC motors. All of them have their specific control card and power amplifier. In the case of the PUMA [the arm] the original linear amplifier was modified and interfaced to the standard control card on board a PC. The head and hand joints are controlled through a bank of switching amplifiers (PWM). Each control card has a DSP on board and to some extent they can be programmed to generate the desired control strategies. For example the head is controlled with a high gain controller while for the arm we employed a low-stiffness control schema.

Encoder signals are collected by the same control cards. In some cases also analog data is read directly from the motor control boards.

Images are provided by standard CCD color cameras and they are sampled at full frame rate by frame grabbers sporting the common BT848 chipset. The original images are sub-sampled as early into the processing as possible to the desired resolution and format (within the Babybot always in log-polar format). Auditory signals are sampled at up to 44 KHz by a standard sound card. The signal coming from the microphones is amplified and conditioned appropriately before sampling. Tactile sensors have their own microcontroller and AD converter. Digital values are sent to a PC though a serial line. Hall-effect analog signals are sampled by yet another card with a bank of AD converters.

The hardware is somewhat heterogeneous since it evolved from previous implementation of the Babybot. Control cards have different CPUs, sampling rates, DSP and software interface. The same applies to the set of PCs where the hardware is interfaced to. They range from older Pentium to the latest generation PIV. Presently the robot is controlled by 14 machines connected via two separate 100Mbit Ethernet networks. One network is totally dedicated to control signals, the other mostly to visual processing.

4. OS independent software interface

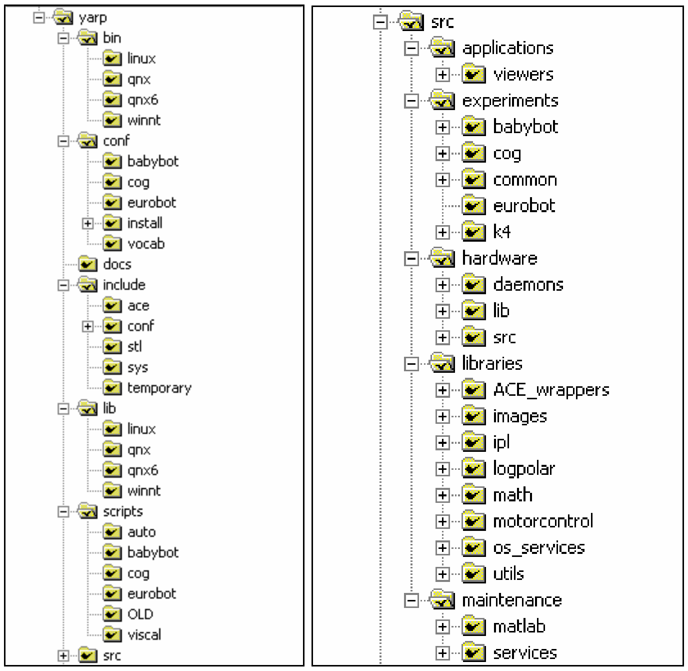
Broadly speaking, the lowest level software components should aim at encapsulating as much as possible the details of the communication and computing layer. The hardware interface should be as seamless integrated into computation as possible. And, finally, most of the code should easily run on different operating systems depending on the requirements of the specific installation. The language of choice in our case was C++. Development tools are Microsoft Visual Studio for Windows and *gcc* for QNX and Linux.

For the task of encapsulating the operating system we naturally relied on existing software. In particular we found convenient to base our implementation on ACE, an open-source library that among many things provides a tiny object-oriented OS wrapper. For more information about ACE please refer to: http://www.cs.wustl.edu/~schmidt/ACE.html.

ACE runs on Windows, Linux, and QNX that were also our target operating systems. Basing our implementation on ACE allowed clearly running all our code on any of these operating systems. From our point of view ACE provided a common C++ class interface for the communication code and the OS wrapper. Advanced ACE functionalities were not fully exploited. We preferred to take a minimalist approach and rely on the minimum subset of ACE that allowed solving our tasks. Following the open-source philosophy, our software was made freely available on SourceForge:

http://yarp0.sourceforge.net/

The most part of the communication code is profoundly inspired [and recycled] from a previous version developed at MIT (Paul Fitzpatrick. **From First Contact to Close Encounters: A developmentally deep perceptual system for a humanoid robot**. PhD thesis at MIT, 2003) and was tested extensively on the humanoid robot Cog on QNX4.25. The latest implementation has been completely rewritten (using ACE) but it maintains the same high level interface. The distribution is not completely user friendly yet as the only way of obtaining the code is through the CVS interface provided by SourceForge. The library in its entirety has been called YARP (Yet Another Robot Platform).



```
yarp
  bin
    linux
    qnx
    qnx6
    winnt
  conf
    babybot
    cog
    eurobot
    install
    vocab
  docs
  include
    ace
    conf
    stl
    sys
    temporary
  lib
    linux
    qnx
    qnx6
    winnt
  scripts
    auto
    babybot
    cog
    eurobot
    OLD
    viscal
  src
```

```
src
  applications
    viewers
  experiments
    babybot
    cog
    common
    eurobot
    k4
  hardware
    daemons
    lib
    src
  libraries
    ACE_wrappers
    images
    ipl
    logpolar
    math
    motorcontrol
    os_services
    utils
  maintenance
    matlab
    services
```

Figure 7 YARP directory structure. On the right the expanse of the source code directory (src).

At the moment of writing, YARP's main directory tree consists of the various branches shown in Figure 7. This includes room for the multiple OS binaries, libraries, and .h files. Among other things the library is completely scriptable which allows an easily run-time configuration of different parts and of different experiments.

The communication code is a C++ templatized set of classes contained in a library called OS_services (see Figure 7 on the right). The main abstraction for inter-process communication is called a "port". A port template class can be specialized to send any data type across an IP-network relying on a set of different protocols. Depending on the protocol different behaviors can be obtained – the implemented protocols include TCP, UDP, MCAST, QNET[1], and shared memory. A port can either send to many target ports or receive simultaneously from many other ports. A port is an active object: a thread continuously services the port object. Being an active object allows responding to external events at run time, and for example it is possible to send commands to port objects to change their behavior. Commands include connecting to another remote port or receiving an incoming request for connection and since all this can be done at run-time it naturally enables connecting/disconnecting parts of the control system on the fly.

Figure 8 shows an exemplar structure of the port abstraction. Each port is, in practice, a complex object managing many communication channels of the same data type. Each port is potentially both an input and output device although for simplicity of use only one modality is actually allowed in practice. This is enforced by the class definition and the C++ type check. Each communication channel is managed by a "portlet" object within the main port. Different situations are illustrated in Figure 8: for example an MCAST port relies on the protocol itself to send to multiple targets while on the contrary a TCP port has to instantiate multiple portlets to connect to multiple targets. In cases where the code detects that two ports are running on the same machine the IP protocol is replaced by a shared memory connection. In Figure 8 a special portlet is shown: this is indicated as "command receiver". As already mentioned its function is that of receiving commands to connect, disconnect, or generically operating on the port. Further ports can run independently without blocking the calling process (if desired) or they can wake up the calling process on the occurrence of new data. In some cases synchronous communication is allowed (TCP protocol).

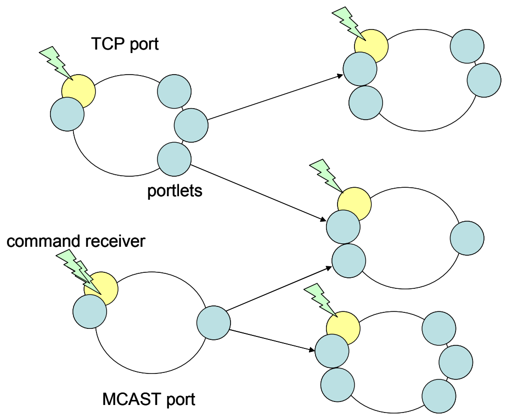


Figure 8 The YARP communication architecture.

[1] QNET is the native QNX network protocol (message passing). It is very efficient and tuned for real-time performance.

Protocols can be intermixed following certain rules. Different operating systems can of course communicate to each other. QNET protocol is an exception and it is only valid within a QNX network.

YARP communication code leads to a componentization of the control architecture into many cooperating modules. The data sent through port can range from simple integral types to complex objects such as arrays of data (images) or vectors. Thus controlling a robot becomes something like writing a distributed network of such modules.

In addition, YARP contains supporting libraries for mathematics and robot type computation (kinematics, matrices, vectors, etc.), image processing (compatible with the Intel IPL library), and general purpose utility classes. We also designed a few modules based on existing Microsoft technology to allow remote controlling Windows machines (this support comes naturally on QNX). In short, these scriptable modules complete seamlessly the architecture allowing the design of scripts to bring up the whole control structure and connect many modules together.

As an aside lately a Matlab interface to ports has been implemented. This allows building Matlab modules (e.g. .m files) that connect to the robot to read/write data. There are basically two advantages: i) complex algorithms can be quickly implemented and tested relying on Matlab existing toolboxes, ii) an additional level of scripting can be realized within Matlab. Matlab provides a relatively efficient and easy to use display library that can be used to visualize the functioning and performance of an ongoing experiment.

In summary, Figure 9 presents schematically the link and dependences between the YARP libraries.

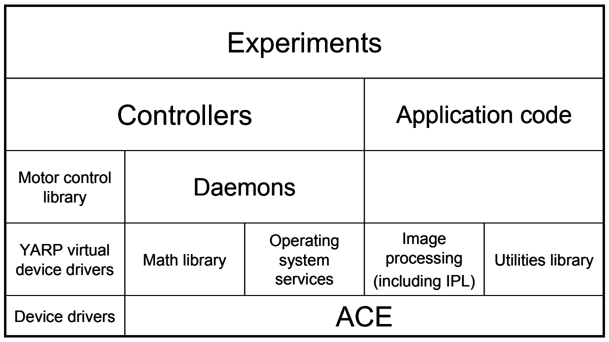


Figure 9 YARP libraries: dependence chart.

5. Robot independent code

One of the goals in writing our control architecture has been that of simplifying the programming of a complex robotic structure such as a humanoid robot. As described in section 3, control cards come in many different flavors and programming them is usually painful. It would be much better if a standardized interface were provided. It would be even better if a suitable abstraction were available.

To solve the first problem we defined a “virtual” device driver interface into YARP. To solve the second, we encapsulated the control of parts of the robot (head, arm, frame grabbers, etc.) into a standardized template class hierarchy.

In short, the virtual device drivers bear much of their structure from the UNIX device drivers. Each card’s driver class contains three main methods: Open, Close, and IOCtl. The latter is the core of the interface. Each device accepts a set of messages (with parameters) through the IOCtl call. Each message accomplishes a specific function. Two different control cards supporting roughly the same commands can be easily (as it was done in our setup) mapped into exactly the same virtual device driver structure, although clearly the implementation might differ.

The next layer is a C++ hierarchy of classes which through templates includes both the specification of the controlling device driver (e.g. the head is controlled through a certain control card) and the idiosyncrasies of the particular setup (e.g. wiring of the robot might differ, or initialization might require different calibration procedures). This hierarchy is shown in Figure 10.

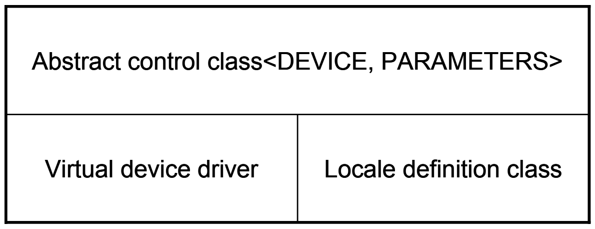


Figure 10 The structure of a control class for a generic device.

6. Robot specific interface

The real "communication" with the robot is carried out through a set of binary modules that use the device driver structure described in section 5. Module customization is at this stage accomplished through configuration files. In the YARP language these modules are called daemons (a term borrowed from UNIX). The daemons directly interact with the remainder of the robot software through YARP ports and in general they export very specialized communication channels. For example the frame grabber has an output port of type "image" and the head control daemon an input port that accepts velocity commands. There are no specific restrictions on the type of ports exported by a daemon since any type of state information about the modules might be required.

Further, some of the daemons accept or send commands of a special type that are generally used to communicate status information. A bus structure based on the MCAST protocol has been implemented to transmit and receive these special messages (called "bottles"). YARP bottles may contain any type of data or even a group of heterogeneous elements of different types. The structure contains identifiers to properly decode messages and interpret the data. YARP bottles create a network within the network of behaviors to realize a high-level control and coordinate a large number of modules.

For example, YARP was already configured to control three different setups: the data-glove acquisition setup, the Babybot, and a second robotic setup at LIRA-Lab where we are investigating aspects of real time control and predictive behaviors.

7. Learning architecture

This layer describes an arrangement of YARP modules that tends to repeat across our robotic architecture. This is not formally into YARP proper but simply an implementation of a particular experiments relying on YARP libraries. Conceptually it forms a layer where to build more sophisticate experiments since for example it provides simple motor control and sensorimotor coordinative behaviors. Overall they could be seen as very high level commands that support positioning, gazing, reaching for visually identified objects, and grasping them.

Grossly speaking, autonomous learning requires a slightly different approach from classical supervised paradigms where data is presegmented and simply fed into a function approximator. Autonomous learning is perhaps closer to reinforcement learning in that it requires action and proper behaviors (exploratory) to gather the training set. Necessarily our architecture will require bootstrapping behaviors supporting building the training set. The question of how much explore and how to get quickly to a solution is an open one in reinforcement learning and unfortunately reinforcement learning itself tend to be difficult, requiring a very large number of samples. In addition, in the case of a real robot we shouldn't allow "spurious" or random control values to get to the low-level controllers; at the basis of any control strategy we should probably have a reasonable "safe" explorative procedure and certainly not a complete random one. Self-supervised procedures can be identified (similar in spirit to feedback error learning) and given the appropriate amount of exploration they can quickly approximate the desired sensorimotor coordination pattern.

When data samples are available in sufficient number with respect to the size of the parameter space of the function approximator of choice the system can start learning and using what has been learnt up to date; necessarily in the long run the influence of explorative behaviors should be reduced. At least two possibilities exist here: learning could be implemented either in batches or fully online. The specific strategy is mostly a function of the algorithm and specific implementation of the function approximation. Inhibition or a functional equivalent should take care of reducing or mixing up exploration with actual "exploitation" of the acquired behavior.

Our discussion is only focused here on the function approximation problem since a good part of the sensorimotor behaviors can be actually well implemented by mapping sensory values onto motor commands or the opposite or even by a combination of the two (e.g. feedback error learning or distal learning).

Another constraint on the design of explorative behaviors is that they should mostly "explore" the space that will be used in the future. Needless to say that failure to do so might result in very poor performance.

The learning algorithm can be conceptually divided in two parts: the one providing the "learning signals" sometimes called the "critic", and the one doing the behavior called the "actor". This distinction is important in motor control problems since the actor must be extremely fast and should work in a small delay regime. On the other hand, the critic could take even seconds or minutes to process the training data and provide infrequent adjustments to the actor's parameters. We maintained as much as possible (apart from trivial cases) this distinction within our system. This division is to some extent compatible with biological mechanisms of learning being these for example the rates at which synaptic changes and growth processes develops in the brain compared to actual spikes' travel times.

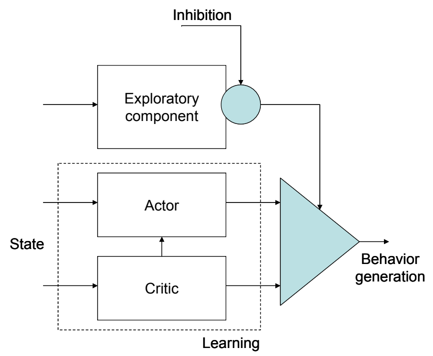


Figure 11 A module for learning sensorimotor coordination

Figure 11 sketches the modules required for each actual behavior acquisition. At the moment of writing we have only conducted a few experiments with the combination and definition of modules presented here. Examples of explorative components are (at the moment) bounded random behaviors (used when training the hand localization map) or early muscular synergies connecting and generating activations of muscles spanning different joints and even different limbs. In learning reaching, these synergies can be exploited to bias the exploration space and avoid random movements. Whenever learning relies on multiple cues, such as visual and motor, having an initial coordination (although imprecise) can be advantageous. One net effect would be the reduction of the learning space that needs to be explored before getting to a reasonable behavior. This strategy was used in our previous work (see G.Metta, G.Sandini and J.Konczak. *A Developmental Approach to Visually-Guided Reaching in Artificial Systems.* Neural Networks Vol 12 No 10 pp. 1413-1427 (1999)).

The actor and critic modules in our experiment consisted of a simple batch learning backpropagation neural network. Although, not the best, it proved to be very reliable so far. Backpropagation has been extensively tested and its behavior very well characterized in literature. Consequently, it is much easier to understand especially when things do not go as expected. The implementation maintains the separation of actor and critic to the point of having a slow batch learning method as critic, and a distinct process providing the behavior. Naturally, given the overall robot architecture, the two modules can be even running on two different machines.

Inhibition and the control of activation and coordination of many behaviors is still argument of further research and no definite implementation has been reached yet. Figure 12 shows the combination of many blocks of this type. In this case too, the realization is completely hypothetical since testing has not been performed yet.

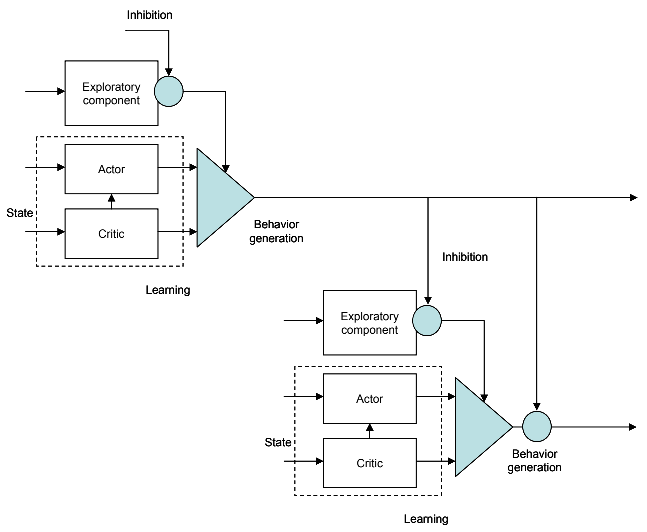


Figure 12 The combination of learning modules in an hypothetical subsumption arrangement.

8. Specific experiments

Based on YARP we started putting together the infrastructure required to support object manipulation on the Babybot setup where the final demo of Mirror will be implemented. Some of the ongoing activity is described in the following section.

8.1. Learning a visual model of the hand

One way to solve the inverse kinematics problem is to learn the functional relation between the head fixation point and the arm posture. This mapping can be used to program a ballistic movement of the arm toward the point in space the head is fixating at. A closed loop mechanism may still be required to compensate for small errors in case the ballistic motion is not accurate enough. In both cases the robot must be able to track the arm end point (the hand in this case) and to distinguish it from the object it is going to grasp. Self-knowledge emerges early on in humans and it is built from the beginning of infant development. Combined double touch and multimodal correlation allow babies to find out that their body is a unique entity in the environment. By moving their limbs around babies learn that when their hand touches their face they feel a synchronous tactile stimulation on both hand and face. The same does not happen when they grasp an object or touch the floor when they crawl. Besides, infants coordinate information from different perceptual systems to disambiguate between parts of the visual field whose motion matches the one they expect based on their proprioceptive and kinesthetic feedback; they know that these parts of the visual fields are likely to be part of their own body.

Correlation between wrist motor commands and motion in the visual field was used in this experiment to segment the hand. The robot performed a periodic movement of the wrist in order to produce a small movement of the whole hand. Visually, a simple motion detection algorithm was implemented computing time difference information between each frame and a model of the background. For each pixel in the "motion image", a zero-crossing algorithm was used and a periodgram computed. Similar period information is computed on the motion signals coming from the motors (or the issued motor commands for what matters). Clearly, pixels that moved periodically, and whose period of oscillation matches that of the motor commands can be selected as being part of the hand. Parts of the image that moved uncorrelated (different period or aperiodically) could be segmented out (i.e. someone walking in the background).

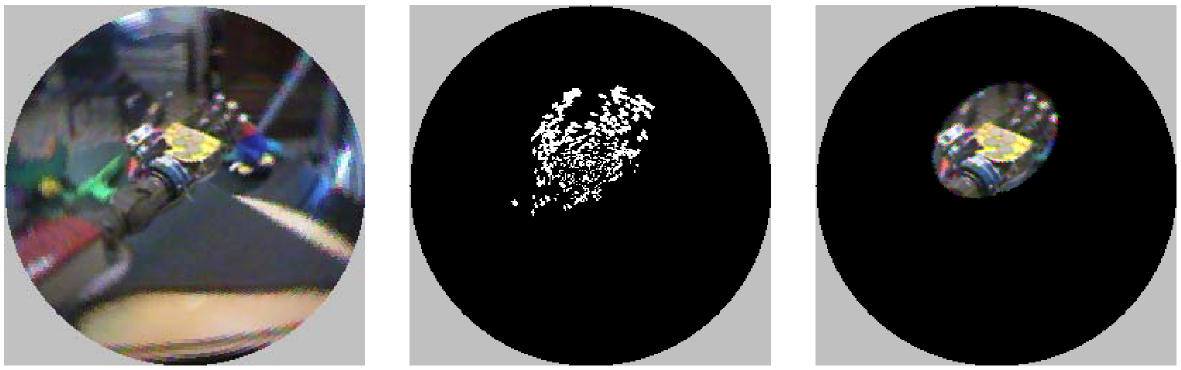


Figure 13: From left to right: image at the beginning of the detection procedure, result of the detection algorithm, segmented image.

The output of the algorithm is a pixel map; in order to get a dense segmented region a series of low-pass filters at different scale is run on the pixel map image. Further, a simple threshold was applied after filtering to get the region of interest. Figure 13 shows examples of the result of this processing.

As it is, this algorithm cannot be used to track the hand of the robot or to localize it during a grasping action. Nevertheless it is a good starting point for building more complicate model of the hand. In particular a simple color histogram was evaluated and a neural network trained to predict the position of the hand in the image plane based on the current arm posture obtained from proprioceptive information. Training data was collected by repeatedly moving the arm at different spatial positions and by running the segmentation. The color histogram was computed in the Hue-Saturation space from many results of the motion-based segmentation. Here we exploit the fact that the body is invariant with respect to the environment and that eventually the background contribution cancels out. Figure 14 shows the result of this process.

The input of the neural network are the arm joint angles, the training sample is the center of the segmented image in the image plane (x,y). In principle the latter should take into account also the position of the cameras (e.g. the head posture). As a first simplification the cameras were kept stationary. After training the neural network computes the expected position of the hand from the arm posture; no visual information is used in this process.

The backprojection of the color histogram detects those parts of the image that have higher probability to belong to the hand. The proprioceptive information predicts its expected position. The latter can be helpful to disambiguate between zones of the image that have the same probability to be part of the hand, or to speed up the search in case a more complex model is used instead of the histogram (see Figure 15).

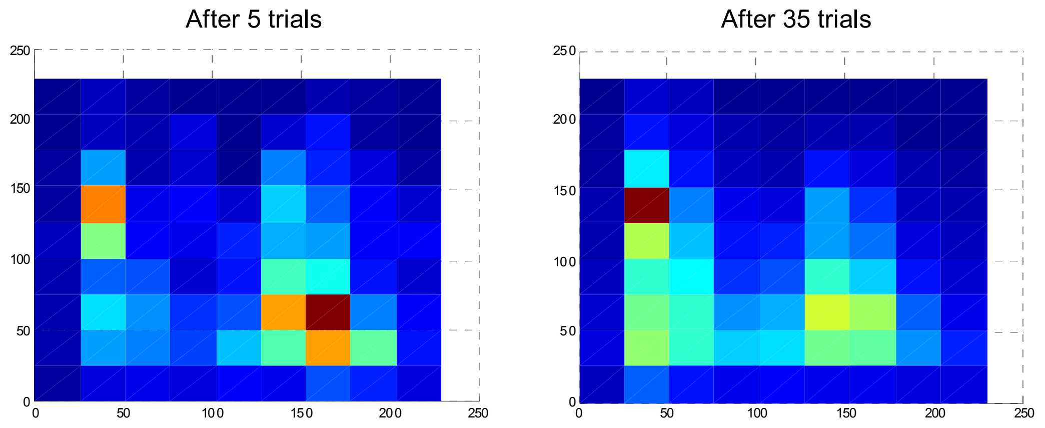


Figure 14: Hand color histograms. The histogram was evaluated in the Hue-Saturation space by using 10x10 bins. The colors map goes from dark blue (0) to dark red (1). After 35 trials the maximum peak moved to the value of the hand's palm (roughly 42 and 125 respectively of hue and saturation).

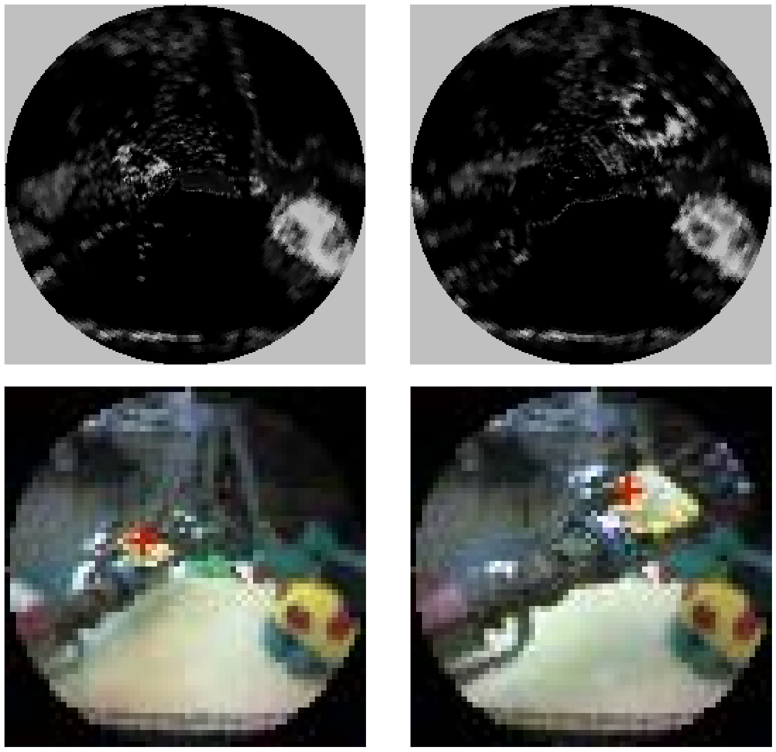


Figure 15: Hand localization. Top: histogram backprojection, brighter pixels have higher probability to be part of the hand. It is not possible to disambiguate between objects that have the same color (the palm of the hand and the car). Bottom: original images, the red "cross" mark here represents the expected position of the hand as computed by the neural network from the current arm posture (no visual information is used in this case).

8.2. Learning gravity compensation

Before actually grasping an object the robot must be able to reach for it, executing a motor action with the arm so that the hand is brought close to the target object. To solve this problem the robot has to compute the position of the object in some reference frame (i.e. the retinal coordinate system) and transform it into a motor command suitable to drive the arm joints (i.e. motor torques). In robotics this problem has been solved in different ways. One solution may be to first solve the inverse kinematics, computing the final posture of the arm in joint space and then compute the torques necessary to drive the manipulator in that particular posture. If the robot acts in an unstructured natural environment during the learning phases it is impossible to avoid unwanted collisions with objects. In this case a low impedance control can reduce the risk of damaging either the arm or the objects the robot is interacting with. Low impedance control (or low stiffness spring-like behavior) can be obtained by reducing the gain of the PID controllers that usually compute the torques required to keep the arm in a desired position. This is more effective if the gravity load terms due to the arm's weight is known *a priori* and compensated for by a feed-forward term. The resulting control loop in this case is shown in Figure 16.

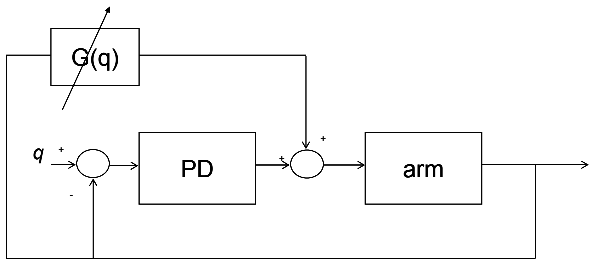


Figure 16: Arm control loop. G(q) is a function computing the torque to compensate the arm's weight from the current arm posture. The closed loop control (PD block) generates the torque required for the motion.

The gravity terms vary with the configuration of the manipulator. In the case of the PUMA arm the feed-forward control is a scalar function of three variables (q1 q2 q3). To simplify the learning the controller uses only the following two variables:

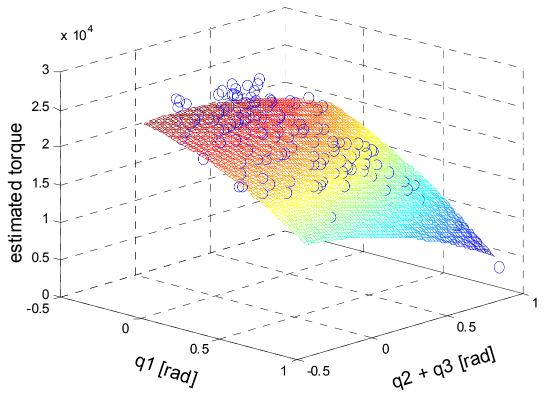
$$\begin{cases} q_1 \\ q_v = q_2 + q_3 \end{cases} \tag{2}$$

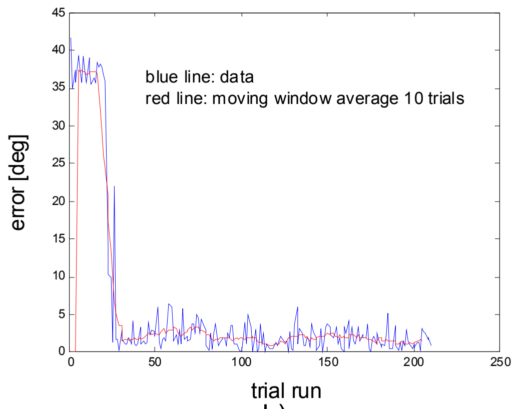
The functions whose parameters have to be estimated have been chosen empirically to be of the following form:

$$G\left(q_1, q_v\right) = aq_1^2 + bq_1 q_v + cq_1 + dq_v + eq_v^2 \tag{3}$$

The learning takes place as follows. At the beginning the feedforward controller does not contribute to the general motion, and only the low-gain PID control is employed. As a result the control is not precise and the arm works under a large error condition. However, every time the manipulator stops in a particular posture (joint configuration) the controller can measure the current torque and use it as an estimation of the gravity term for that particular arm posture. This measure is fed to the controller and it is used as a training sample to the feedforward model. After a certain number of learning steps the gravity compensation is activated and its output added to the output of the PID. If the forward model is precise enough the PID is no longer needed to keep the arm in a particular posture and the position error is consequently small. In spite of the above standing simplifications, no force feedback is available in the PUMA arm and the output voltage of the control board was used instead (meaning that the frictional terms were not properly considered). The PID controller is still required but with lower gain.

Figure 17 shows the error during the learning phase and the gravity term for different arm postures. Only the joint relative to the shoulder is reported here because it is the one that is supporting more weight.





a) b)

Figure 17: ***a) Gravity load torque for the arm shoulder. Blue circles: sample points; surface: approximated function. b) Performance of the controller during online learning. The error drops very quickly after 25 trial runs when the feedforward term is activated.***

8.3. First grasping strategies

Neonates reach for objects but, they do not grasp them during this action. The reason is that reaching and grasping are coupled into extension and flexion synergies and, therefore, it is difficult for the child to flex the hand while the arm is extended. If an object is put into the hand, however, at other occasions the neonate might grasp it. Grasping is performed with the whole hand. Only much later children start employing relatively differentiated finger movements (8-9 months). The ability to grasp objects at a very early age constitutes an important means for interacting with the environment. For this reason it seems reasonable to implement the same mechanism in the robot. Similarly to what happens in newborns, tactile stimulation of the palm, initiates a clutching action with index, medium, ring, and small fingers (the thumb is not used in this action).

Force sensing resistors (FSRs) are mounted on the hand to give the robot tactile feedback. These commercially available sensors exhibit a change in conductance in response to a change of pressure. Although not suitable for precise measurements, their qualitative response can be used to detect touch and measure to some extent the force exerted to the object surface. Five sensors have been placed in the palm and three in each finger excluding the small finger (see Figure 18).

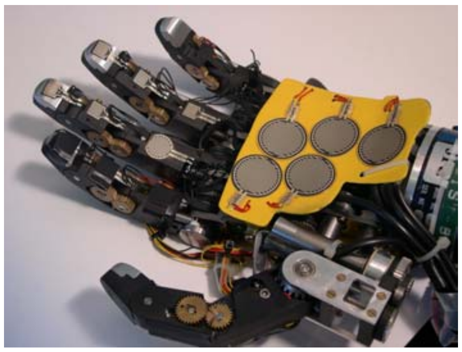


Figure 18: Tactile sensors. Five sensors have been placed in the palm and three in each finger excluding the small finger (the sensors in the thumb are not visible here).

Figure 19 shows data recorded during a grasp elicited by tactile stimulation. At time T1 a soft ball touches the palm (upper trace) eliciting a motor response. The lower trace here reports one of the encoder of the index finger. The finger touches the ball at time T2 and continues pressing it until time T3; the object is held between fingers and palm from T3 to T4. At time T5 it falls off the hand. Although still qualitative these plots show the proprioceptive information that can be gathered by this simple grasping action.

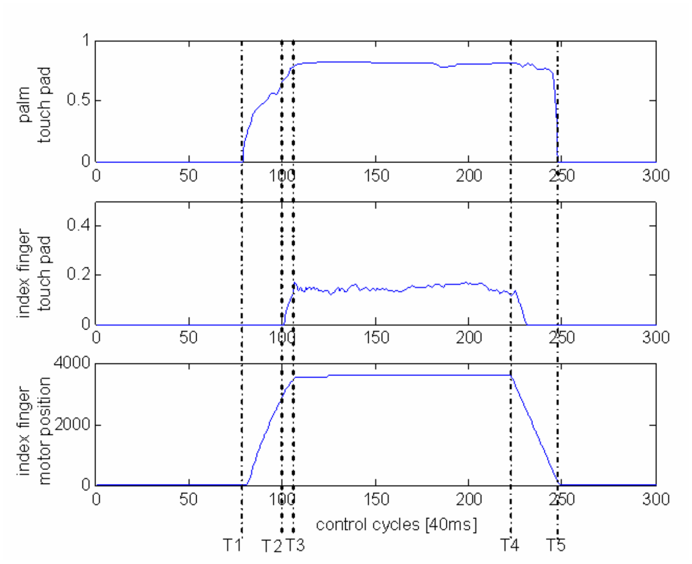


Figure 19 Data recorded during a grasp. Upper and middle: palm and index touch sensor signals, normalized arbitrary scales. Below: index finger motor encoder, first phalanx. The scale in this case is encoder ticks; the conversion factor being 0.015 deg/tick (4000 corresponds to 60 degrees). See text for details.

Newborns reveal a larger variety of finger movements involving both the whole hand and differentiated finger movements. They may be used in exploring objects but not when grasping them. At this age, however, patterns of finger movements are only loosely related to the child's interaction with the environment. The patterns seem to a large extent be the spontaneous expression of a vocabulary of movements just like cooing is the spontaneous expression of speech sounds that are not yet functionally connected into larger functional units (Rönnqvist and Von Hofsten: Neonatal and Arm Movements as Determined by a Social and an Object Context, Early Development and Parenting, Vol.3 (2), 81-94 (1994))

We are aware that more articulated synergies are required for the robot to explore the use of skillful grasps on different objects. It also is important to stress that the implementation of this touch elicited grasp action was very useful to test the performance of the hand and the tactile sensors in a complete closed loop scenario.