UNIVERSITY OF TWENTE

HUMAN FACTORS & ENGINEERING PSYCHOLOGY

MASTER'S THESIS

Bringing the iCub to life with neural network activation

Author:
Janina Roppelt
s1194526

First Supervisor:
Prof. Dr. Frank van der Velde
Second Supervisor:
Prof. Dr. Ing. Willem Verwey

June 30, 2016

Abstract

Robots are getting more and more sophisticated and involved in our daily lives. With growing complexity of tasks to fulfill, the importance of technical cognition is on the rise. The field of technical cognition does not only enhance robot behavior but can also help us to further our understanding of the human brain. The robot discussed in this thesis is the humanoid iCub. The iCub is developed to impersonate a toddler, both in physical and psychological terms. One of the most important characteristics of a human is his efficient and sustainable learning. There are numerous artificial neural networks that try to simulate the learning process of the brain. This thesis uses the neural network architecture of the motor-blackboard. The theory is discussed and implemented in a simulation, which is then integrated to work with the iCub as input and output device. It is shown that the motor-blackboard architecture provides a good basis for a sustainable learning program. Further research should be done on how values have to be set and how rhythms affect human movements.

Contents

1 Introduction

"Human beings have dreams. Even dogs have dreams, but not you, you are just a machine. An imitation of life. Can a robot write a symphony? Can a robot turn a canvas into a beautiful masterpiece?"

Sonny: "Can you?"

The quote above is from the 2004 movie "I, Robot". Sonny is an intelligent robot that developed a consciousness and fights to stay alive. This film is only one of many art pieces that star intelligent machines. Another example is "The Hitchhikers Guide to the Galaxy" where a prototype of the Genuine People Personalities technology, called Marvin, develops severe depressions and boredom because of his high intellect. Or "The Matrix", where machines became so intelligent and powerful that they can deceive people to believe in a safe world, while feeding on power generated by the humans' bodies.

Robots are a quite controversial technical development, not at last because of science-fiction novels that sometimes glorify but mostly vilify intelligent machines. Though some people are still afraid that machines will take over the world, robots are already a reality and facilitate our modern lives. Think of production lines for example. Dull, monotonous and possibly dangerous tasks don't have to be performed by humans anymore. But robots are getting more advanced. Today there are machines that can lift heavy weights and thereby protect humans from injuries (Grey & Joo, 2014; Chu et al., 2014). There are drones that can be used in areas that are of high importance, yet inaccessible to humans, for example due to radiation (Cho & Woo, 2016). On a more personal level, robots are used to treat people with mental illnesses, like children with autism (Pennisi et al., 2015; Yun, Kim, Choi, & Park, 2015; Mohd et al., 2014).

So what about the evil robots feared by many people? One theory that tries to explain why humans are uncomfortable with some robots while accepting others is the uncanny valley. When the appearance of objects becomes more human, there is a point where people feel inherently uncomfortable about the humanness of still inhuman objects. Examples for this are zombies or corpses. (Mori, MacDorman, & Kageki, 2012) The uncanny valley is an important phenomenon to consider when building human-like robots.

Using the humanoid iCub, Sciutti, Rea, and Sandini (2014) discovered, that the importance of different robotic attributes depends on the age of the person interacting with a robot. While younger children prefer human looking machines, teenagers and adults are more attracted by functionality. An explanation for this might be the expectations one has when interacting with a robot.

It is a fact that robots become more human-like, both in appearance and concerning the tasks they are involved in. Hence, situations in which robots are used become more and more advanced. As a consequence, machines have to execute more and more complex tasks. Though humans might execute such tasks without noticeable effort, they are difficult to implement on machines. In essence, one has to build brain activities from scratch. Indeed, the main problem when programming a robot to act like a human is the lack of knowledge on how humans function on a micro level. Due to the evolving natures of technology and psychology, the cooperation with machines is an essential tool in understanding the human brain. Simulating neural networks is one way of researching learning in detail. At the same time, simulations can serve as a sound basis for machine learning, which has the ultimate goal to increase their efficiency and usefulness.

This thesis will focus on the neural architectures of sequence learning with the humanoid iCub as robotic basis. The outline of the thesis will be as follows. Section two introduces the iCub and describes the motivation to use it. In section three, the theoretical framework of learning in general, and sequence learning in particular will be laid out. Also, artificial neural networks in this area will be discussed. Section four is a description of the motor-blackboard model used as architecture for this thesis. In section five, the implementation of the motor-blackboard, as well as the integration with the iCub, are explained. Section six discusses the findings and usefulness of the model, as well as limitations and further research with the motor-blackboard simulation. Section seven and eight conclude the thesis and suggest future work on the iCub.

Summary of the goals:

- Learn more about human learning by simulating the theoretical framework of the motor-blackboard.
- Integrate the simulation with the iCub to enable the robot to learn.
- Give interested students the knowledge and skills needed to start experimenting with the iCub themselves.

2 The iCub

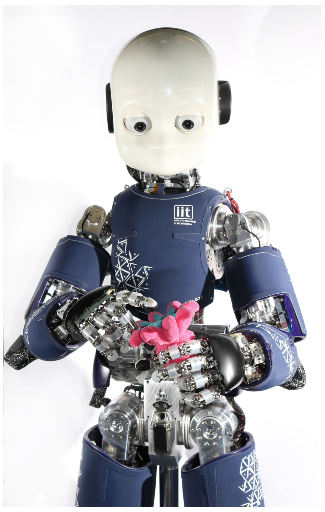


Figure 1: The iCub playing with a stuffed animal

"The iCub is the humanoid robot developed at IIT as part of the EU project RobotCub and subsequently adopted by more than 20 laboratories worldwide. It has 53 motors that move the head, arms & hands, waist, and legs. It can see and hear, it has the sense of proprioception (body configuration) and movement (using accelerometers and gyroscopes)." (www.icub.org)

The intentions behind building the iCub were to create an open source project, both in hardware and software. Embodied cognition could be studied on the one hand. On the other hand, more knowledge and understanding of natural, as well as artificial, cognitive systems could be generated. RobotCub argues that manipulation plays an essential role in the development of cognitive capabilities, which consequentially means that people are not born with many of their basic skills. Those skills are rather developed in early childhood, for example through interaction with other human beings. This is why the iCub should in the end impersonates a toddler of around 2.5 years, in size as well as in cognitive abilities. While the size is already given, the cognitive abilities will need further research. The amount of time and manpower needed to develop the cognitive paradigms is one of the reasons why the iCub is an open source project. (Metta, Sandini, Vernon, Natale, & Nori, 2008; Tsagarakis et al., 2007)

Since the release of the iCub robot, a diversity of projects has been launched. There are physical projects that are mainly about controlling the motors in a meaningful way. One example for such a project is balancing the iCub in order to make it stand in every position. This opens possibilities to execute tasks where the robot has to move (Liu & Padois, 2015). On the other hand, there are projects that are concerned with perception and cognitive functions like decision-making. Ramirez-Amaro, Beetz, and Cheng (2015) for example let the iCub perceive human behavior whereby he has to make decisions about what he is experiencing. The iCub is also used for social experiments. An important topic is acceptance of robots by people. Gaudiello, Zibetti, Lefort, Chetouani, and Ivaldi (2016) tested acceptance through trust. They found that the iCub, though looking rather human, is more trusted in functional than social tasks. To peruse suchlike experiments, software is essential. The programs that construct the iCub environment are discussed in the next section.

2.1 The iCub software

Besides the software running on the iCub itself, there are several programs to facilitate communication with the iCub. The communication of the robot is managed by a YARP (yet another robot platform) server. This server registers the iCub, as well as the programs used to operate it. It then leads all communication to the desired component of the iCub netwerk.

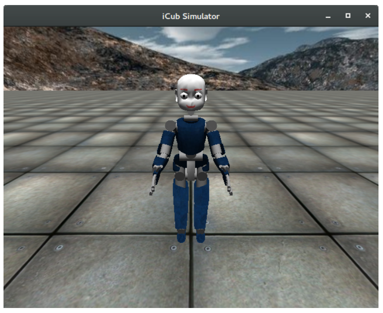


Figure 2: The iCub simulator

As the iCub and all of its software is written in C++, communication between programs and the iCub in this language will be straight forward. To facilitate communication with other languages, like Python and Java, bindings have to be installed. These bindings are functioning as a translator between program out- and input and server out- and input.

Next to the YARP server, the iCub software also includes a simulator. The iCub simulator shows an on-screen simulation of the robot and can be used to test programs in an effective and secure manner before playing them on the iCub. This is not only to protect the robot, but also an appropriate way to correct minor errors. The simulator can also be used with self-generated objects for the iCub to interact with.

Running a program via YARP is not the only way to control the iCub (simulator), the YARP-motor-GUI can be used, too. For every motor in the iCub there is a joint that can be manipulated. This will have immediate impact on the simulator. Using the GUI is especially useful if one wants to visually experiment which joint manipulation will result in which movement. It can also help to set input values in a program, as values can be easily matched to movements through observation.

Appendix A includes a tutorial on how to install all necessary iCub software and how to start communicating with the iCub through the motor-GUI and Python.

2.2 The iCub at the University of Twente

In march 2016, the iCub arrived at the University of Twente. The robot is to be placed in the DesignLab, a creative and cross-disciplinary ecosystem aiming to connect science and society through design (www.utwente.nl/designlab/). All interested students from different studies should be able to start projects on the iCub themselves. Considering the variety of students and their technical knowledge, it is seen as important to have a low beginning hurdle. This is why Python is chosen as programming language for this project. Being a high level programming language that is easy to read, Python is not only a good beginners language to learn, but also integrated in the study program of Psychology. Psychology students are a main target group, especially when interested in Cognitive Psychology and Ergonomics, and they normally have a bigger gap to close when starting to work with technology.

At the moment only a C++ tutorial exists, written by students of the study group RAM (Robotics and Mechatronics). Though the practical parts of their work might be difficult to understand, readers that want to learn more about the structure, components and communication channels of the iCub are encouraged to read their report. (Jager & Meijering, 2015)

3 Motor sequence learning and neural networks

This section introduces learning with the focus on motor sequence learning. After discussing the literature on the sequence production task, a task where people are trained to reproduces sequences of movements, neural network architectures for this task are introduced.

3.1 Learning in general

Everything in life has to be learned. From being an infant until we die, we are constantly learning new things which help us through life. Without being able to learn we would not be able to live.

Everything we do as human beings involves usage of our brains and is based on movements. There are movements necessary to keep our bodies alive, like the contractions of the heart muscle, and movements that help us to stay healthy and functional, like blinking. There are movements that help us to achieve our needs, like chewing. Lastly, there are movements that help us to achieve the goals we have in life. (Kalat, 2009)

As the iCub is a robot, the movements important for it to learn are the ones that are needed to achieve a goal. The goals we have change during our lifetime. Baby's learn, amongst other things, to move their body through crawling and walking, and to make sounds. They essentially learn basics every human need to master before learning more complex skills like jumping, singing or debating.

While basics skills are learned naturally, mastering a complex motor skill will not only take time and practice, but especially discipline. For a gamer to become professional, it will take hours a day, for years, until his fingers react as fast as possible to the situation on the screen. As will an Olympic athlete who strives to perfect his movements, to make them as effective as possible.

The iCub, being an indoor type, sports are not the best option to learn. On the other hand, common computer games are a bit too complex, as a lot of cognition is needed, and playing one might be to brutal for the innocent little guy. It was chosen to facilitate skills the iCub would need in order to learn a musical instrument. A rather "simple" music instrument to begin with is the piano. Though he might be too young for the desired starting age of around four to eight years, depending on who you ask, simple melodies should be no problem. This is not because the piano is an easy instrument to master, but because pressing keys is less prone to errors when compared to instruments like a violin or trumpet. It has the advantage of being less stressful for the ears of the learner's parents, too.

Learning how to make music has different components. The first one to be learned is usually a feeling of music and rhythm. Infants begin to develop a sense of musical phrasing between 4 and 7 months after birth. With around 6-7 months, they can distinguish musical tunes based on rhythmic patterns. At the end of their first year, they can recognize a melody even when played with other notes. (Berk, 2009) After developing a musical sense, toddlers can learn to make music themselves. Children of around 2-3 years are taught to sing and play rattles.

Before being able to learn a more complex instrument, the conventional way is to learn how to interpret written music. Learners need to know how rhythm translates to note-times and pauses, and which position of the hand on the instrument corresponds to which note. In the piano example, that means knowing which note is played with which key. The part of giving meaning to written music is cognitive learning rather than motor learning and will not be discussed in this thesis.

Figure 3: A toddler on the piano

The last step is to actually learn to play the instrument. Learning a music instrument is traditionally done through following explicit instructions (sheet music) as well as repeated practice. Learning strategies like implicit or explicit recognition of patterns as well as trial-and-error may also be used but are less dominant. Based on these strategies, a sequence of movements is learned until it can be performed in a seemingly automatic manner. This is referred to as motor sequence learning. (Abrahamse, Ruitenberg, de Kleine, & Verwey, 2012)

3.2 Discrete sequence learning and chunking

Sequential structures are the building blocks of most, if not all, actions that are done to achieve our goals in everyday life. These structures allow us to perform learned actions with limited mental effort and without constant monitoring. Constant learning throughout our life is enabling us to act automated and to learn more and more complex skills. (Abrahamse et al., 2012)

Discrete sequence production tasks (DSP) are seen as a manner to study the sequence structures which are the building blocks of complex behavior. (Abrahamse et al., 2012) Decades of research have shown that after learning, chunks of sequences are loaded into a short term motor buffer. It was shown that most chunks have a length of around four. Chunking can be seen in the DSP through reaction time. The execution of the first movement in a sequence takes significantly longer than the following ones. Another observation was that after four movements reaction time rises again, but it will be lower than the initial reaction time. This discovery led to the introduction of the idea of concatenation, which means combining chunks with each other to create a bigger chunk. Though concatenation also involves loading, it is quicker than initialization. The presented explanation for this is that initialization involves general processes which only have to be done once every sequence. (Abrahamse et al., 2012)

Think of the example of playing the piano again. Most songs have more than four notes and can, if learned to perfection, be played in one piece with only one (conscious) initialization. Of course, a longer song or a whole composition might have more initializations, for example at the beginning of every couplet or melody part.

Abrahamse et al. (2012) discussed that initiation and execution of sequences are two different processes, meaning that "playing" a melody with another finger combination will take longer to execute, but the initialization time will be the same. A reason for this might be that execution in such a case involves more cognition than in the "normal" case. If you are at a party and want to give drinks to your friends, the execution is easy to master, but if they changed places, and you have to cross your arms, a less common movement has to be made which will most likely take longer to conclude.

The discrete sequence task showed that in order to develop skills no structural knowledge of the sequence is necessary. (Abrahamse et al., 2012) This fact makes is possible that the iCub can learn and execute movements without the need to establish good cognition first. In relation, Abrahamse et al. (2012) argues that the most likely representation of sequential movement in our brains is in slowly developing motoric codes. This representation in joint angles and forces is one of several representations. Another representation is the faster learning effector-based coordinate systems (Hikosaka et al., 1999; Panzer et al., 2009). The motoric codes representation is more likely, as it is more efficient in controlling movement execution. Conveniently, though the iCub could also be coded in a coordinate system based manner, the easiest and straightforward way is to use motoric codes.

3.3 Cognitive and motor processor

There are two distinct processors in our brain that guide movements. The cognitive processor is used whenever cognition is needed to move correctly and the motor processor when movement is guided automatically. So the cognitive processor is controlling movements, while the motor processor is executing them. (Abrahamse et al., 2012)

The distinction between processors can explain why activities in our brains are different in early learning stages than in later ones. In early stages, the cognitive processor is dominant. Through learning, the motor processor slowly becomes the dominant one. The accuracy of the idea of two processors can be strengthened through analyzing behavior that occurs when one of them is needed but not present, the moment in which we make mistakes.

Take a password you use frequently as an example. At some point you are asked to change your password due to security reasons. You change your password and proceed to log into the service. At the moment you want to type your new password, your favorite song comes on the radio and your attention shifts to the music. When your attention is back, you recognize a warning that you've entered an old password. What happened is that the cognitive processor got distracted by the song, which left the motor processor on the fast track. The cognitive decision to use the just edited password did not occur, because the cognitive processor was distracted, and the motor processor did what it was taught to do.

As far as the iCub goes, this is a welcome observation. With cognitive and motor processors being two separate systems, they can be implemented independent of each other. The only important thing is that they have a clear protocol for communicating. As stated earlier, this thesis will mainly focus on the motor processor functionalities.

3.4 Modes of learning

There are three different modes of sequence learning: the reaction mode, the associative mode, and the chunking mode. (Verwey, 2003, 2010) In reaction mode, actions are controlled through cognition, as a reaction to an external stimulus. In our example, that is playing with notes as primary source. In chunking mode, the learner will be able to produce sequences automatically, the piano equivalent being able to play without sheet music.

In the associative mode, the other two modes are combined. There is some automation, but only if the sequence is played slow. In order to be able to quicken it up, external stimuli will be necessary. Take an intermediate level learner that wants to improve his piano skills. He might be able to play some parts "from memory", but is using sheet music in order to minimize mistakes and be able to play quicker as there is less cognitive remembering involved.

3.5 Neural networks

Motor learning is one of the, if not the, most important function of our brain. Unfortunately, due to the unconscious manner of learning, there is little exact knowledge on how our brain does learn exactly (Abrahamse et al., 2012). In order to understand and be able to reproduce brain activity, artificial neural network simulations are used.

Neural networks are networks of neurons that are connected via dendrites in such a way that neurotransmitter communication between them gives meaning to a higher-order process. Simulating these biological neural networks via hardware and/or software, can help with understanding the possibilities and limitations of such a communication network. On the other hand, it can be used to make artificial intelligence more realistic, an outcome that is desired for the iCub. Additionally, the brain is a very efficient computer, especially when it comes to learning and adapting, so rebuilding it might help to advance computing. There are numerous artificial neural networks that try to simulate their biological counterpart. There is not one good model, but rather models that are better than others for a specific task.

3.6 Neural networks for motor tasks

Clements (2015) discusses different neural networks attempting to model the discrete sequence production, namely the model of Hélie, Roeder, Vucovich, Rünger, and Ashby (2015), the Leabra framework (Gupta & Noelle, 2007) and the TELECAST model (Hélie, Proulx, & Lefebvre, 2011).

In the model of Helié, automatic sequence production is situated in the cortical area, while sequence knowledge is based in the supplementary motor area. In the early stages of learning, the basal ganglia activate a motor plan in the supplementary motor area, which produces a response within the cortical areas. The learning process is done by Hebbian learning (Hebb, 1949). Hebbian learning occurs when one cell is firing and thereby contributing to the firing of another cell. If this happens frequently, the connection between the two cells is strengthened, and the relationship which might have begun accidentally will become causal. Automation is gained after extensive training. At that moment, the role of the basal ganglia is taken over by a sub-net of units of cortical-cortical connections within the supplementary motor area, that represent the sequence. The weakness of the model is that it does not include a visual loop (Clements, 2015).

The Leabra framework is situating motor learning in two parallel neural networks, moderated by a cognitive control mechanism. The separate networks are called controlled path and automatic path. While the controlled pathway acquires a declarative representation of the sequence, the automatic pathway encodes procedural representations, which takes longer. When the automatic pathway has gained high proficiency the cognitive control mechanism will increase its contribution. On the other hand, when mistakes are made, the contribution of the controlled pathway will be higher. The framework can account for generalization between tasks, fast recovery in re-learning, and complete automation of sequence production, while not being able to stop over-learning and simulating the difference in reaction time between controlled and automatic sequence production. Another shortcoming is that the neural underpinnings are not confirmed yet. (Clements, 2015)

The TELECAST model is a cognitive bottom-up model of explicit learning. Similar to the Leabra model, output is generated by processes, an explicit and an implicit one. Explicit knowledge is modeled by a Bayesian belief network, implicit knowledge by an unsupervised connectionist network. As in the model of Helié, learning is done via Hebbian learning and practice. This model does not include neurobiological details as it is namely about rule discovery in humans. (Clements, 2015)

The architecture chosen for this thesis is the motor variant of the neural blackboard theory (van der Velde & de Kamps, 2006; van der Velde, 2016), a framework consisting of different modules that allow independent learning and usage. This independence facilitates fast, efficient and durable learning. It also addresses the problem of most neural networks that when changing a single component, everything will have to be learned again. The next section will discuss the development of the architecture and how the neural blackboard theory was adapted to motor learning.

4 Motor-blackboard theory

The neural network theory was initially proposed to solve the four problems of Jackendoff (2002) for neural instantiation of combinatorial structures in cognition. According to Jackendoff, these problems do not only arise in language processing, but in all combinatorial cognition. To solve these problems, the neural blackboard architecture proposes different processors that work on one blackboard without interfering with one another (van der Velde & de Kamps, 2006).

Within the growing research on technical cognition, the focus of artificial neural networks is shifting from biological correct models of the brain to rule discovery in humans and neurally inspired networks as basis for concept based computing. The neural blackboard is developed as such a model, supporting the idea of in-situ concept based computing. In-situ means in the present. With the combination of concept based computing, it supports the notion that concepts used in processing and production are not copied or transported but used from the position they are in. In robotics, in-situ computing is promising as it can be implemented parallel and with neuromorphic hardware. Both reduce the time and power needed to process complex information which is crucial to make robots feasible to use. (van der Velde, 2016)

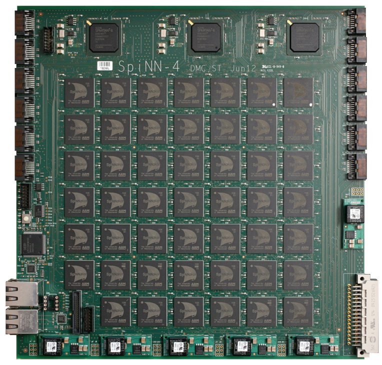


Figure 4: Neuromorphic hardware: A 48-node SpiNNaker board, equipped with 864 ARM cores (running at 200 MHz) and 6 GByte RAM (128 MB x 48 chips). 48 cores can run at least 48 processes in parallel.

In-situ computing using a neural blackboard architecture is proposing a solution to ambiguity. While the used concepts of a task are ambiguous in nature, this is resolved by dynamic competition of possible representations during processing. (van der Velde & de Kamps, 2015) This results in the advantage of such an architecture, that components can be reused. They only have to be in the structure once, which in turn means that they only have to be learned once, too. Through processing stored information in an in-situ way, unnecessary overhead of learning, as well as use of valuable memory, is prohibited.

The motor-blackboard is a variant of the combinatorial blackboard architecture adapted to motor sequence learning rather than higher cognitive processes like language and reasoning. In this thesis the motor-blackboard architecture will be put to test as a basis of learning the iCub how to play the piano. While becoming a pro musician is a long term goal, the motor-blackboard will facilitate the motor learning part. Specifically, that adds up to the iCub being able to learn sequences of movements given a rhythm and feedback. Furthermore, this program will enrich the theoretical model with functionalities like rhythm learning, saving information and bringing learned associations into action. It has to be noted that the scientific focus lies on the motor-blackboard itself. Other components are implemented without thorough scientific consideration and should therefore be subject to further research.

4.1 General architecture of the motor-blackboard

The motor-blackboard model (van der Velde, 2015) has four main components: chunk nodes (C-nodes), sequence nodes (S-nodes), gates or gate columns, and fingers. The components can be learned and used independently from each other, meaning that every component, once learned, can be used by the others without additional effort. Take a rhythm for example. Once learned, a person can use it with his right hand, but also with the left hand or any other part of the body. Using components independently makes learning efficient, as nothing has to be learned twice. The relations between the components are illustrated in figure 5. In the following sections, the different components and their functionality will be discussed.

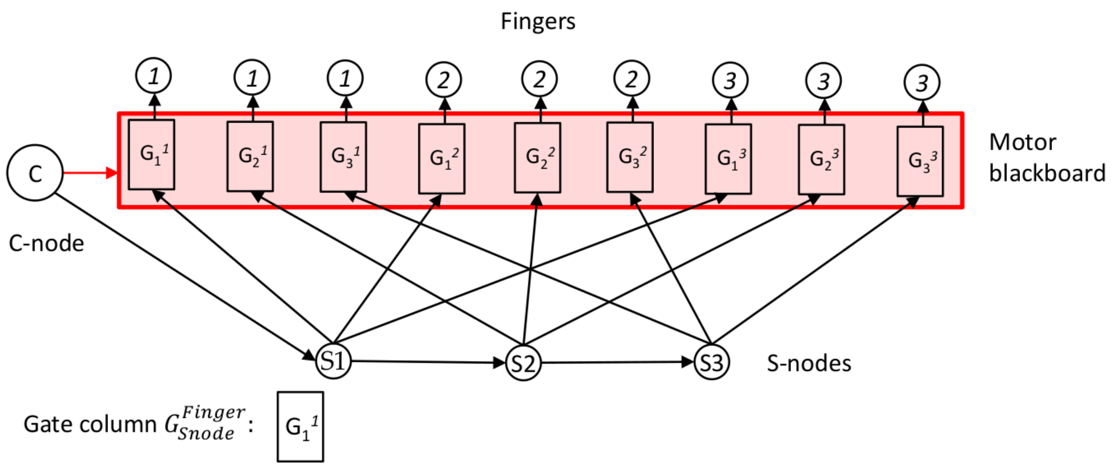


Figure 5: Architecture of the motor-blackboard. The red rectangle symbolizes the blackboard, consisting of the gate columns. A C-node interacts with every gate in such a blackboard.

4.1.1 Fingers

The fingers are, as the name suggests, representatives of real fingers. In the figure on the architecture, there are three fingers. The reason why all of them are shown three times, is that every finger is connected to more than one gate. There are as many gates as fingers times the number of S-nodes.

The model can theoretically be used for nearly any amount of fingers. Though at first it might seem illogical to use any number above ten, that might not be the case in practice. Fingers are an abstract idea in the model. They are receivers and senders of neural activation. While the receiving part will lead to an action, the sender part gives feedback on that action. So why should there be more than ten? Think of the piano again. A chunk will learn which finger to move at which point in time, but a piano has more than ten keys that can be played. So if a chunk only learns which finger to move, there would still be a lot of cognition involved. It is more likely that there are as much “fingers” as notes that can be played. This implies that the blackboard does not send the signal to move to a finger itself, but rather to some sort of translator that knows which exact movements to make when a specific key on the piano should be played.

Such a translator could also be expressed as a motor-blackboard, but on a lower level. It would be less conscious, as the movements which have to be combined are learned early in life. It would also be less straightforward which rhythms should be use. We can hear music, but we are not consciously experiencing how long our movements take and which components of our bodies are used exactly.

For implementing such a motor-blackboard, a low level programming language like C should be used. In comparison with Python, it is way faster and close to hardware, which is exactly what such a process would need. After all, the hardware of a robot is closely related to the body of a human, as its software is to the humans’ mind. As programming the translator as a motor-blackboard would require a thorough understanding of a low level language and hardware, this thesis uses a simplified version that does not stand up to the model. Though not implemented now, the independence of the components will allow such a translator to be added later without having to make changes to the present program.

4.1.2 C-nodes

C-nodes are the chunks that store which finger has to be used at which position in the sequence. One could think of chunk nodes as telephone numbers that have to be learned. How longer the number, how harder it gets to remember it in one part. At some point the learner will split the information, consciously or unconsciously. As discussed in the theoretical framework of the discrete sequence learning section, the ideal chunk number is around four elements per chunk, and long sequences are subject to concatenation, the combination of several small chunks to learn longer sequences. In case of the telephone number concatenation would be necessary to remember a whole number and not just a part of it.

A C-node in figure 5 could look like: “1,2,3” or “3,1,2”. The numbers here are one of the three fingers displayed. In the beginning of the process, when no C-nodes are learned yet, the knowledge of which finger to move must come from a cognitive process. Later on, that knowledge comes from the learned association between the C-nodes and gates. How that works will be explained in the gate section.

4.1.3 S-nodes

S-nodes are responsible for the timing of sequence execution. They are activated by a rhythm that is determined in the initialization. This rhythm will dictate when, and how long a S-node has to be active. The rhythm does have to be activated exactly once in the beginning of a movement, thereafter it runs independent of the C-nodes.

When an S-node is active, it activates specific gates which, in combination with the C-node activations on the gates, leads to the right finger activation. Before discussing this process in detail, the next section will explain how a gate column is designed.

4.2 Architecture of the gate columns in the motor-blackboard

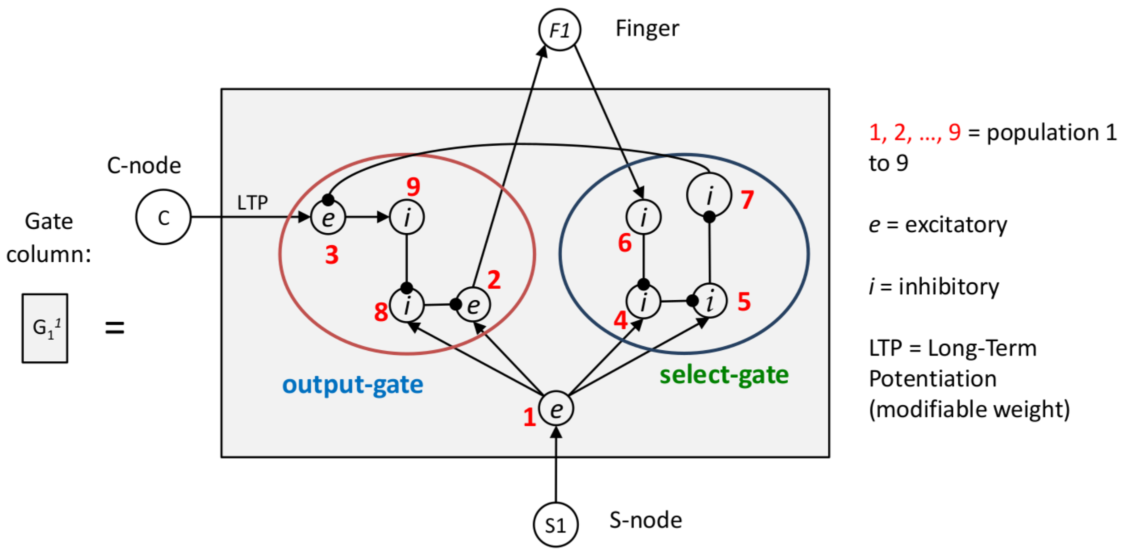
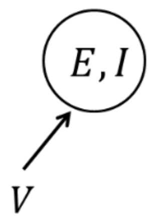


Figure 6: Gate column positioned between the first S-node and finger number one.

Figure 6 shows the architecture of a single gate, including the three other components of the model. Such a gate could for example be implemented on one of the 48 chips shown in figure 4. Every gate holds nine Wilson-Cowan (WC) populations. WC populations exist of excitatory and inhibitory neurons which are assumed to act as groups (Wilson & Cowan, 1972). This structure has several advantages. First of all, such a population is not dependent on one neuron. This is useful as neurons are sensitive. If learning and movements would depend on a single neuron, a simple thing like a drink or a ball hitting a head could leave a person with serious damage to his capability to live. Secondly, a population is quicker than a single neuron could be. While a single neuron can fire 50 times per second, a population of 100 can fire up to 2000 times per second. Thirdly, such an architecture is less prone to errors. One malfunctioning neuron that is firing too little or too much will not have a great impact on the outcome, as the outcome will be an average of the whole population.



Population:

- Combination of excitatory (E) and inhibitory (I) cells.
- External input (V)

Wilson & Cowan (1972):

$$\tau_E \frac{dE}{dt} = -E + f(\alpha E - \beta I + V(t))$$

$$\tau_I \frac{dI}{dt} = -I + f(\gamma E - \delta I + V(t))$$

$$f(x) = f_{max} \frac{1}{1 + e^{-\delta(x-\theta)}}$$

$$V(t) = input$$

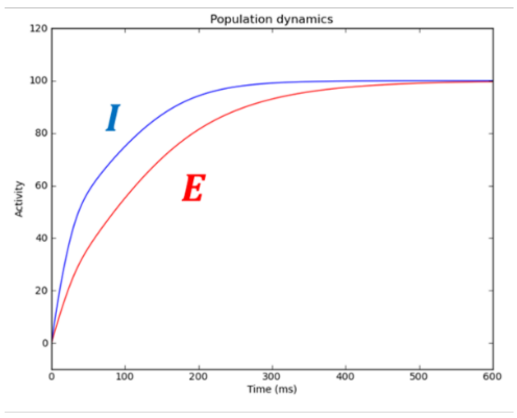


Figure 7: Wilson-Cowan equation. Inhibitory cells are activated faster than excitatory cells to prevent endless activation.

According to Wilson and Cowan (1972) every population has inhibitory as well as excitatory populations. How the activations of these groups within a population are calculated is shown in figure 7. A population is either excitatory or inhibitory depending on which neurons produce the output of the populations. Excitatory populations activate other populations while inhibitory populations do the opposite, they inhibit the activation of another population. From the nine populations in a gate, three are excitatory and six inhibitory. The population which has to be activated in order to activate the gate as a whole is the excitatory pop1. This population is activated by S-nodes and activates pop2, pop4, pop5, and pop8.

4.2.1 Select gate

The select gate consists of pop4, pop5, pop6, and pop7. Pop7 is constantly activated from an external population and therefore constantly inhibits pop3 in the output gate. When the gate is active, thus pop1 is activated by an S-node, pop4 inhibits pop5, which will lead to no changes in the activation between pop7 and pop3.

4.2.2 Output gate

In the output gate, pop3 is constantly inhibited by pop7. If the gate is active, pop1 activates pop8 and pop2. Pop2 would normally activate the finger but is inhibited by pop8. This ensures that S-nodes alone cannot activate fingers.

4.2.3 Feedback

In the early learning stages, fingers will be activated through cognitive processes. In case that happens, pop6 will be activated, too. That leads to pop4 getting inhibited. If pop4 is inhibited, pop5 is activated due to the pop1 activation. As pop5 is an inhibitory population, it will inhibit pop7, so that that the inhibition of pop7 on pop3 is omitted.

4.2.4 LTP

If pop7 is inhibited, pop3 is activated by the Long-Term-potential (LTP) which is a connection between the C-node and the gate through pop3. If pop3 is activated, it activates pop9, which then inhibits pop8. Because the S-node is still active, pop2 now can activate the finger.

If both pop3 of a specific gate and a C-node are activated, the long term potential of this connection grows. LTP is building on Hebbian learning (Hebb, 1949) as discussed in the neural network section. Through Hebbian learning, after some time and rehearsals, the LTP activation on pop3 will be stronger than the inhibition of pop7 on pop3. That leads to the process being started by the initialization of C- and S-nodes alone, without any explicit cognitive actions.

4.3 The learning process

Now that all components are explained, revisit figure 5. Learning occurs when three prerequisites are given. A C-node has to be active, a S-node has to be active, and there has to be feedback of a finger. Given the gate architecture that is all that is needed in order to learn a sequence.

Take the chunk 231 as an example. First, the C-node gets activated, which will also active a rhythm. So the first S-node will active the gates G_1^1, G_1^2, and G_1^3. Then, from a cognitive process, for example from interpreting notes, finger two gets activated and sends feedback to all gates connected to it, in this case G_1^2, G_2^2, and G_3^2. The only gate that is activated by both, the S-node and the feedback is G_1^2, so in this gate the process described above takes place. That process will strengthen the LTP for the activated C-node and that gate.

Next, the second S-node will be activated, activating G_2^1, G_2^2, and G_2^3 in return. This time, the cognitive process will result in feedback on gates G_1^3, G_2^3 and G_3^3, leaving activation and strengthening LTP for gate G_2^3 and the activated C-node. The same will happen for the last S-node. Here gate G_3^1 will be the one to connect to the C-node via LTP.

After some trials, LTP between the C-node on the named gates will be so high, that the activation of the fingers will happen without cognition, as LTP activates pop3 stronger than pop7 inhibits it.

5 Motor-blackboard implementation - from simulation to a responsive iCub program

This section covers the implementation of the motor-blackboard. The program is based on the elementary simulation used in "Outline of a motor neural blackboard" (van der Velde, 2015). The technical report showed that the gate simulation produces realistic reaction times. Theoretically, the motor-blackboard should be implemented on neuromorphic hardware. As such an implementation is time-consuming and costly, this thesis will simulate the network on normal hardware. Such a simulation can be a proof of concept before taking further steps. As discussed earlier, the main programming language used is Python, though calculations are done with a C library. Such a library does increase performance significantly, as calculations are done in binary machine code. There are code examples throughout in the text. The whole source code of the project can be found in appendix B. Note that the examples in the text are to clarify the architecture, rather than being working code. For reusing code, refer to the appendix.

5.1 General architecture

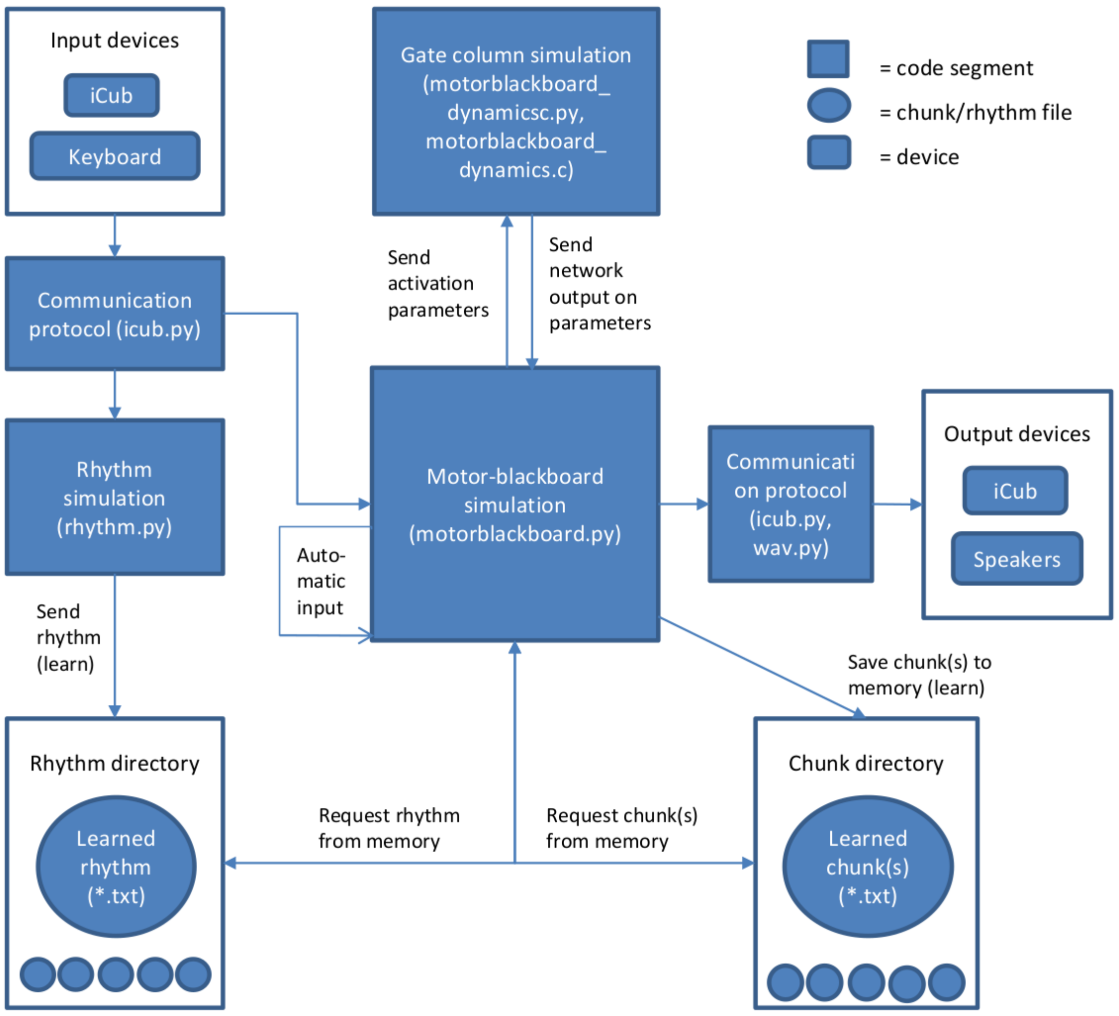

Input devices
iCub
Keyboard
Gate column simulation (motorblackboard_ dynamicsc.py, motorblackboard_ dynamics.c)
= code segment
= chunk/rhythm file
= device
Communication protocol (icub.py)
Send activation parameters
Send network output on parameters
Rhythm simulation (rhythm.py)
Motor-blackboard simulation (motorblackboard.py)
Communication protocol (icub.py, wav.py)
Output devices
iCub
Speakers
Auto-matic input
Send rhythm (learn)
Save chunk(s) to memory (learn)
Rhythm directory
Learned rhythm (*.txt)
Request rhythm from memory
Request chunk(s) from memory
Chunk directory
Learned chunk(s) (*.txt)

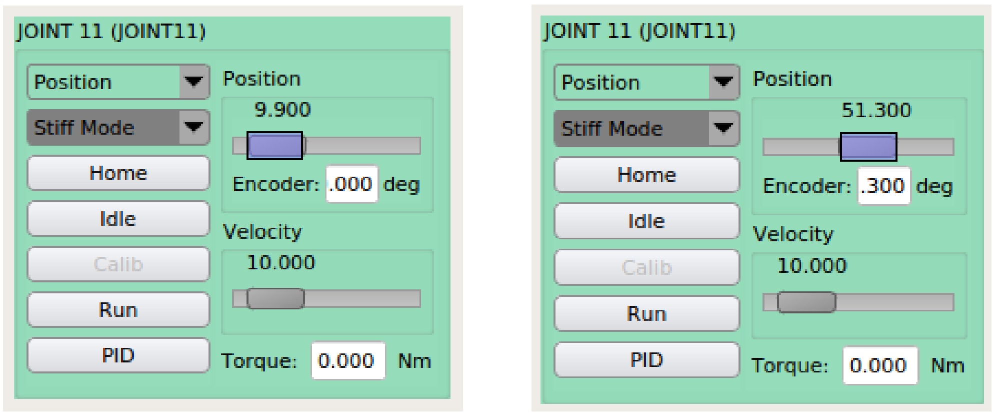
Figure 8: Overview of all components of the programs architecture in terms of software parts, file structure and devices.

An overview of the software architecture is shown in figure 8. The components respond to the motor-blackboard theory. The blackboard itself is implemented in the motor-blackboard simulation. C-nodes and S-nodes are saved in separate directories and will be loaded into the blackboard when needed. Feedback from the fingers is given by input devices, while activation of fingers is sens to the output devices. Activation per gate column will be calculated in a separate simulation which is constantly communicating with the motor-blackboard simulation. An addition to the model was made with the protocol to translate output of the blackboard to iCub actions and vice versa. Such an external translation was necessary to keep complexity in the blackboard itself maintainable, and the components interchangeable.

Technically speaking, there are five scripts. One for learning a rhythm, one for learning a sequence, one to simulate the gates, and two to translate the simulation output into “movements”. Besides that, there are two directories in which “knowledge” is stored, one for the rhythms and one for the C-nodes. Nodes are saved in the language-independent Json format. The reason why Json is used is that it is a lightweight data-interchange format that is easily integrated with Python.

The two programs that can be run are the C-node (movements) simulation and the rhythm simulation. While only the C-node simulation uses output devices, both the C-node and the rhythm simulation use input devices. There are two types of input devices, the iCub and a conventional keyboard. The reason to also use a keyboard as input device is the ease and speed gained in testing. The desired device has to be set before running the simulation. While the rhythm simulation cannot generate automated output, the movement simulation is able to do so. In that case, no input device is needed, but the program will need more parameters to function correctly. It has to be noted that the iCub was not available during the process. All iCub code is therefore tested with the iCub simulator and the YARP-motor-GUI. Figure 10 illustrates how a finger is moved using the YARP-motor-GUI. Although there should be no fundamental differences, handling the program might be slightly different on the actual robot.

Figure 9: Assisting a child with key pressing. This is the general idea on how the iCub should learn these movements.


JOINT 11 (JOINT11)
Position | Stiff Mode
Position: 9.900
Encoder: .000 deg
Velocity: 10.000
Torque: 0.000 Nm

JOINT 11 (JOINT11)
Position | Stiff Mode
Position: 51.300
Encoder: .300 deg
Velocity: 10.000
Torque: 0.000 Nm

(a) The joint in its home position (b) The joint if the finger is “pressed down”

Figure 10: Effect of LTP on pop3 activation

5.2 Rhythm simulation

Before a sequence can be learned, a rhythm to activate the S-nodes must exist. Which of the components of the general architecture are included in rhythm learning is shown in figure 11.

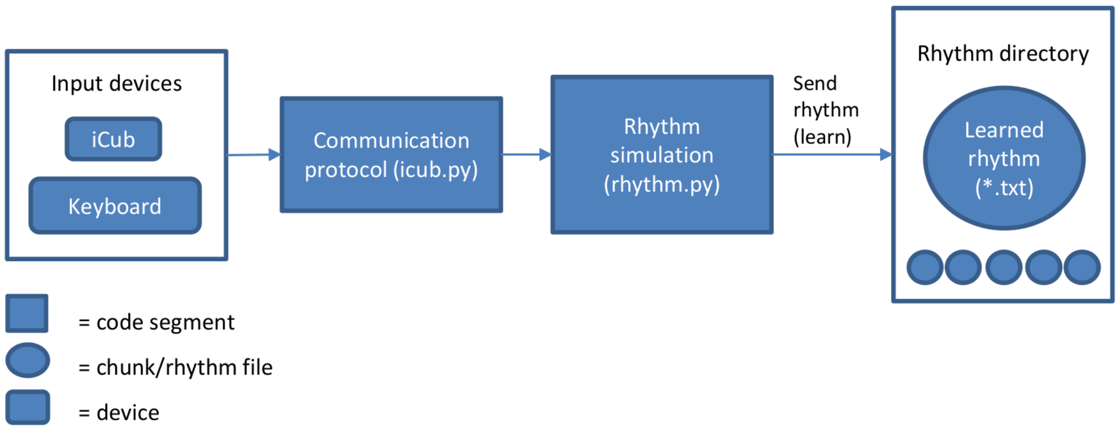


Figure 11: Components used in rhythm learning

The general idea behind this part of the simulation is that rhythms are practiced and refined through intensive learning. Before starting the program, the input device has to be set. Input can be received from the iCub or a keyboard. For the keyboard variant Pygame is used. Pygame is a module of Python that facilitates keyboard and mouse captures as well as the control of output devices like speakers and the monitor.

```
# Input device
# 0 = keyboard
# 1 = iCub
input_device = 0
```

Listing 1: rhythm.py

After the input device is set, the program is ready to run. It will automatically ask the user for a name to be given to the rhythm. That is done to create a reference point that is more than a number and therefore recognizable. The user is asked to decide on the number of positions the rhythm should have and a number of runs, too. Using more than one run is important in order to correct little mistakes made throughout the runs.

```
# Ask for name of the song to learn
song = raw_input('What song should I learn? (name of the song) -->')

# Define how to save the rhythm (song.txt in the S-node directory)
rhy_in_dir = 'Snodes/' + song + '.txt'

# Ask for number of positions
sn = int(raw_input('How many positions does the rhythm have? -->'))

# Ask for number of repetitions
train = int(raw_input('How often should I play? -->'))
```

Listing 2: rhythm.py

Once the variables are set, the program will calculate and establish all necessary numbers and objects. A rhythm is split up into parts when is has more than four nodes. This is done to reduce complexity later on. The number four is chosen based on the findings that a chunk has around four positions, as discussed in the theoretical framework.

```python
'''
Import necessary modules based on input device
'''
if input_device == 0:
    import pygame
elif input_device == 1:
    import icub
'''
Calculate amount of chunks the rhythm should have, based on the amount of positions
'''
amount = (sn - (sn % 4)) / 4 + 1
if sn % 4 == 0:
    amount -= 1

'''
Make a new rhythm array filled with zeros
Use eight positions per chunk -> two values per S-node: the first is the begin time of a
    S-node, the second the end time
Use as many chunks as calculated based on all positions in the rhythm
Append a zero to save how often the rhythm was learned
'''
rhythm = [[0] * 8 for i in xrange(amount)]
rhythm.append([0])
```

Listing 3: rhythm.py

While the general idea of rhythm learning is to save a begin and an end time of every S-node, the exact behavior of the program behind this point is dependent on the chosen input device. The keyboard variant measures for how long a key is pressed. The pressed time is the time a S-node should be activated in the main simulation. With the iCub, it is measured for how long a finger of the iCub is pushed down. Because it was not possible to actually do that on the real robot, the motor-blackboard-GUI was used to simulate the event. As with the keyboard version, the time the finger stays down will be the activation time of an S-node.

Learning works in loops. The outer while loop sets up as many runs as the user requested. The next while loop keeps the program running until a user action is registered. The user action starts the next while loop, which is the actual learning run. This loop saves the begin and and time of a key or finger action. Rhythm is saved as an array that is filled from left to right. The begin time of an action is only put at even numbers of the array, while release times are put at uneven numbers. As the position is increased immediately after the registration of a press or release, and no position can be skipped, the array will fill up neatly. Once a learning run is done, the outer while loop will either start a second run or the program will go through to saving the rhythm. Figure 12 shows how a run of the rhythm simulation looks like.

```python
'''
Set variables to manage loops
'''
# number of values that have to be saved (begin and end time times positions)
s = sn * 2
```

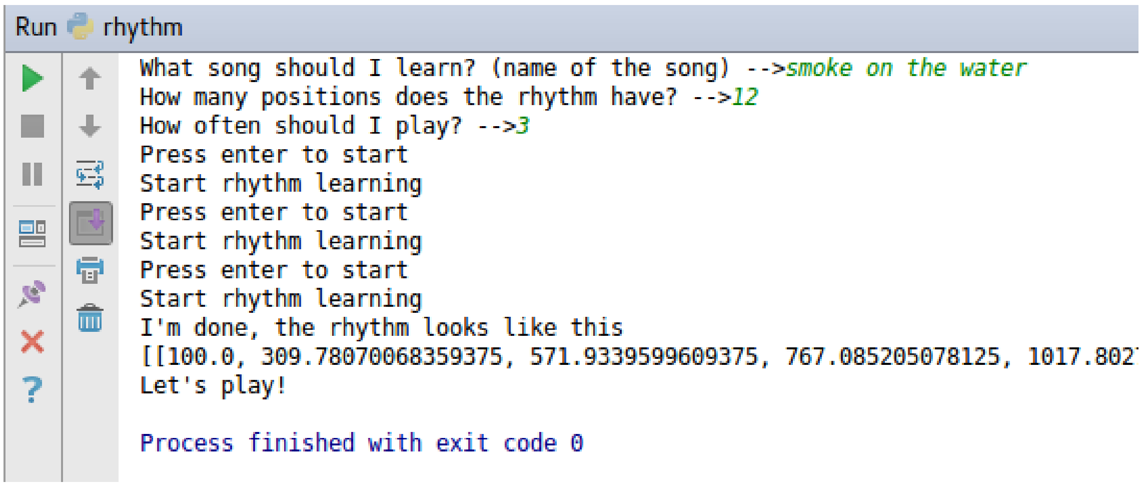
```
t = 0
snode = 0
learn = 0

'''
Initialize pygame if pygame has to be used
'''
if input_device == 0:
    pygame.init()
    screen = pygame.display.set_mode((640, 480))

    # Outer while loop: redo learning loop as long as there are runs left
    # First inner while loop: wait until enter is pressed
    # Second inner while loop: one iteration of learning a rhythm
    while train > 0:
        print "Press enter to start"
        t = 0
        snode = 0
        wait = True
        while wait:
            pygame.event.pump()
            pressed = pygame.key.get_pressed()
            if pressed[pygame.K_RETURN] == 1:
                wait = False

        # Set start time of learning loop as reference
        tim = time.time()*1000

        print "Start rhythm learning"

        # Go on while there are positions to fill
        # Fill rhythm from left to right
        # rhythm[snode] is the chunk of the rhythm that is active
        # rhythm[snode][t % 8] is the position of a chunk the loop is in
        # snode is increased after 8 steps (begin and end time for 4 positions)
        while t < s:
            pygame.event.pump()
            pressed = pygame.key.get_pressed()

            # If space is pressed an t is even, save begin time of S-node
            if pressed[pygame.K_SPACE] == 1 and t % 2 == 0:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time()*1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1

            # If space is released an t is uneven, save end time of S-node
            if pressed[pygame.K_SPACE] == 0 and t % 2 == 1:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time()*1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1
        # increase number of times rhythm was learned
        rhythm[-1][0] += 1

        train -= 1

# If input device is the iCub, do same as above with the difference that not key presses
    are monitored, but the angles of one of the iCubs finger
# icub.get_Pos is called to receive the angles of the joints of the robot
elif input_device == 1:
    while train > -1:
        print "Move the index finger to start"
        t = 0
        snode = 0
        wait = True
        while wait:
            if icub.get_Pos(0) > 15:
                wait = False
        tim = time.time() * 1000
        print "Start rhythm learning"
        while t < s:
            if icub.get_Pos(0) > 30 and t % 2 == 0:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time() * 1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1
            if icub.get_Pos(0) < 10 and t % 2 == 1:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time() * 1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1
        rhythm[-1][0] += 1
        train -= 1

'''
Calculate delta of the first position and normalize array so that the rhythm begins at
    100.
'''
delta = rhythm[0][0] - 100
for x in xrange(amount):
    for y in xrange(8):
        rhythm[x][y] -= delta

'''
Calculate theoretical and actual length of the array in order to calculate where the
    learned rhythm ends
'''
theo = amount * 8
real = s
numb = theo - real
end = rhythm[amount - 1][7-numb]

'''
Fill up the rest of the array with dummy values, 50 higher than the endpoint.
```

```
As they are all the same, no S-nodes will be activated by these dummies (no time in
    between begin and end time)
'''
for k in xrange(8):
    if -delta - 1 < rhythm[amount - 1][k] < -delta + 1:
        rhythm[amount-1][k] = end + 50

'''
Save file to predefined location
'''
with open(rhy_in_dir, 'w') as outfile:
    json.dump(rhythm, outfile)

print 'I\'m done, the rhythm looks like this\n', rhythm, '\nLet\'s play!'
```

Listing 4: rhythm.py

```
# Get positions of joints
# Arguments: finger to get the position of
# Translate finger to joint
# Get all joints, wait until every joint has a value
# Return requested value
def get_Pos(finger):
    joint = finger * 2 + 9
    if joint == 9:
        joint = 10
    pos = yarp.Vector(jnts)
    while not(iEnc.getEncoders(pos.data())):
        t.sleep(0.1)
    return pos[joint]
```

Listing 5: icub.py



```
What song should I learn? (name of the song) -->smoke on the water
How many positions does the rhythm have? -->12
How often should I play? -->3
Press enter to start
Start rhythm learning
Press enter to start
Start rhythm learning
Press enter to start
Start rhythm learning
I'm done, the rhythm looks like this
[[100.0, 309.78070068359375, 571.9339599609375, 767.085205078125, 1017.802
Let's play!

Process finished with exit code 0
```

Figure 12: Output of the rhythm learning simulation. The rhythm of smoke on the water was learned three times.

5.3 Gate simulation - motor-blackboard dynamic

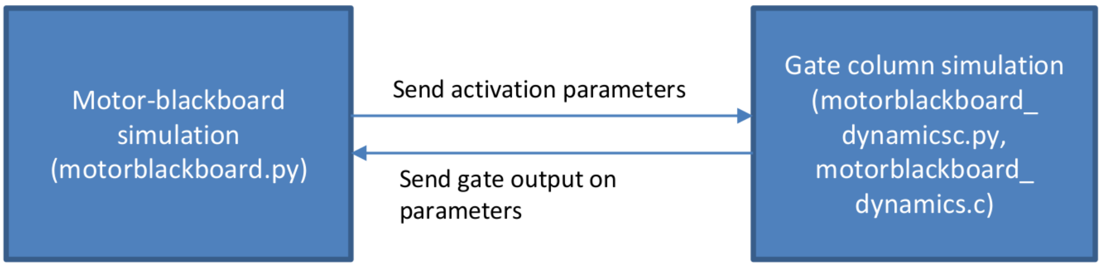

Motor-blackboard simulation (motorblackboard.py)
Send activation parameters
Send gate output on parameters
Gate column simulation (motorblackboard_ dynamicsc.py, motorblackboard_ dynamics.c)

Figure 13: Integration of motorblackboard_dynamics

The activation of excitatory and inhibitory populations is calculated at every step of the motor-blackboard simulation, using the Wilson-Cowan equation explained in the previous section. The inhibitory and exhibitory parts of every population in every gate are calculated. Another function determines which of both outputs is returned per population. That will be the output also affecting other populations within the gate. The output of a population is based on its specific inputs.

```
"""
Calculation of the populations in the gating columns
See "Outline motor model.pdf" Fig 3
Input for a gate column:
Snode_e: input from the excitatory population of the sequence node connected to the gate
Cnode_in: input from the Chunk nodes, determined in the for loop
Fback: feedback from the fingers (actuators in general)
Inh_chunk: global input to give constant activation for pop7
"""
for k in xrange(gate):
    if k < f:
        position = 0
    elif k < 2 * f:
        position = 1
    elif k < 3 * f:
        position = 2
    else:
        position = 3
    Cnode_in = 0.0
    finger = k % f
    Cnode_in += LTP[globc][position][finger] * Cnode_e[i, globc]
    mb.gate_column(i, k, finger, position, Snode_e, Cnode_in, Fback,
                   Inh_chunk, p1_e, p1_i, p2_e, p2_i, p3_e, p3_i, p4_e, p4_i,
                   p5_e, p5_i, p6_e, p6_i, p7_e, p7_i, p8_e, p8_i, p9_e, p9_i)
```

Listing 6: motorblackboard_2016.py

The rules of activation and the calculations are implemented in motorblackboard_dynamicsc.py. Note that the actual calculations are done using a C library (motorblackboard_dynamics.c). This is done to keep the model scalable. With more than three fingers and S-nodes the simulation got too slow to be feasible. Calculating the outputs with C led to an improvement in speed with factor ten. All code of the gate simulation can be found in appendices B.3 and B.4. It is not discussed in depth as it solemnly is an translation to code of the rules within a gate discussed above.

5.4 Motor-blackboard simulation

The motor-blackboard is the most complex part of the program, as it integrates all other components. After discussing the general setting, the integration of components will be addressed in detail. Similar to the rhythm simulation, the motor-blackboard can be used with different in- and output devices. Input can be given via a keyboard, the iCub, and to a certain extend automatically. Additionally, the speed of the simulation can be set. The rhythm gets multiplied with the speed number. That will result in more step calculations per S-node activation. More steps add up to more time for populations to be activated. Highly trained sequences can be played with a speed of one or two. When learning a new sequence, the rhythm should be slower. In that case, a speed value above ten is advised.

```python
'''
Configurations
'''
# Devices to be used:
# 0 = Automatic
# 1 = Keyboard
# 2 = iCub as input
# 3 = iCub as output of automatic simulation
use = 2

# The factor the rhythm is multiplied with
# Simulation is "faster" with a low number
speed = 2
```

Listing 7: motorblackboard_2016.py

Other inputs that have to be changed within the code are the learning factor, the success chance of automated learning, and the begin weights of newly established C-nodes and the gates. They are changed in the because these settings are not likely to be changed often when the program is in a learning cycle.

```python
'''
Configurations for cognition and speed af learning
'''
# Begin weight long term potential
bw = 0.05

# Begin chance to learn right
bcg = 0.7

# How quick the program learns
learning_factor = 1.01

# Activation needed to learn
learning_threshold = 30
```

Listing 8: motorblackboard_2016.py

The simulation is step based. Step based means that the speed depends on time per step. That is why parts of the program are translated to C. When using C with around five fingers and four S-nodes in a rhythm part, a speed of two produced a sequence that was as quick as the learned rhythm, while the same situation in Python makes the simulation unbearably slow.

Once the program is started, the user will be asked to specify the run configurations. These include the song to be played, the number of runs, and in the case of a new song the amount of fingers the program will need to learn the song. Given the information, the program calculates all necessary values and prepares all objects that will be needed for the blackboard to work. As the setup contains a lot of computing it is only presented in appendix B.2.

```
'''
Make run specifications with user input
'''
new = raw_input('Would you like me to learn a new song or play a learned one? (new,
     learned) -->')
if new == 'new':
    LTPU = 1
    songs = raw_input('What song should I learn? (name of the song if a rhythm exists or
    train for training rhythm) -->')
    song = 'Snodes/' + songs + '.txt'
    songc = 'Cnodes/' + songs + 'c.txt'
    fnew = int(raw_input('How many different notes does the song have? -->'))

if LTPU == 0:
    songs = raw_input('Which song do you want to play? -->')
    song = 'Snodes/' + songs + '.txt'
    songc = 'Cnodes/' + songs + 'c.txt'

runs = int(raw_input('How often should I play? -->'))
```

Listing 9: motorblackboard_2016.py

In the code segment above, options are given for songs. These are the three songs the program already knows. In the following, examples will show output on the melody of smoke on the water. Figure 14 shows the activation graph of the rhythm and the chunk nodes. The motor-blackboards handling of both components will be discussed in the following section.

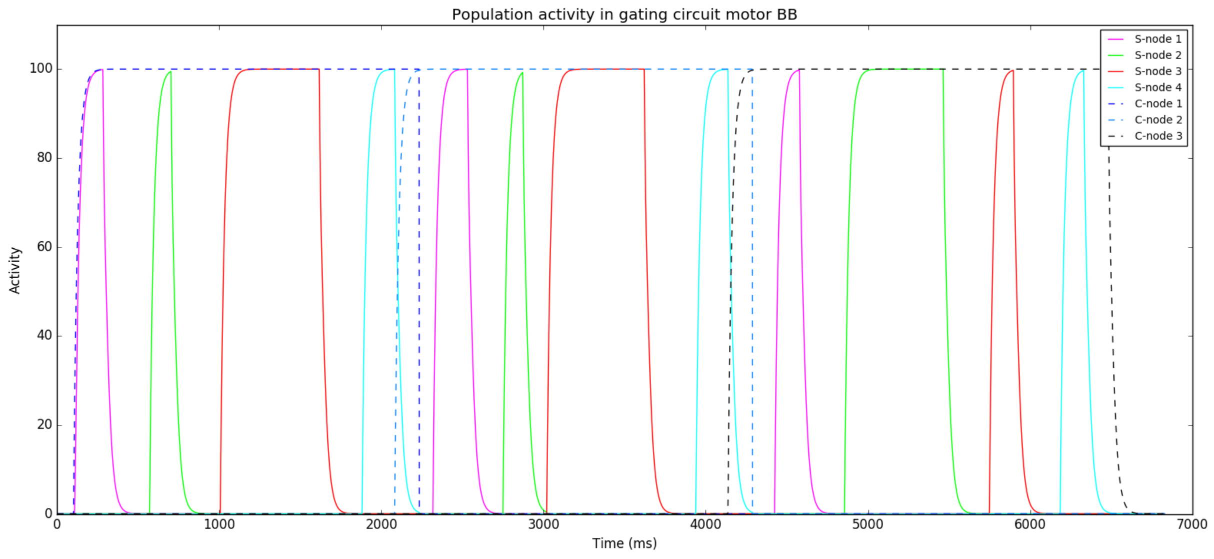


Figure 14: Activation of C and S-nodes in the example of smoke on the water: Three C-nodes with four S-nodes each.

5.4.1 C-nodes and S-nodes

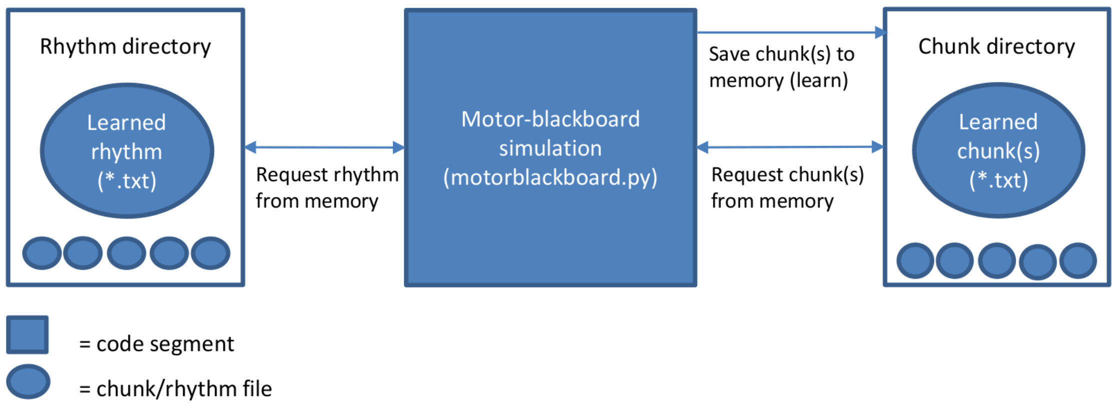

Rhythm directory
Learned rhythm (*.txt)
Request rhythm from memory
Motor-blackboard simulation (motorblackboard.py)
Save chunk(s) to memory (learn)
Request chunk(s) from memory
Chunk directory
Learned chunk(s) (*.txt)
= code segment
= chunk/rhythm file

Figure 15: Schematic integration of S-nodes and C-nodes into the motor-blackboard

Activation and Deactivation of C-nodes and S-nodes Which C- and S-nodes should be used is decided by the user, as described in the previous section. The variable *song* tells the program where to search for the S-node, the variable *songc* where to store the C-node, and in case it is already learned, where to find it.

```
# import rhythm from file
with open(song, 'r') as outfile:
    r = json.load(outfile)


# import LTP from file if already learned
if LTPU == 0:
    with open(songc, 'r') as outfile:
        LTPL = json.load(outfile)


# export LTP to file
with open(songc, 'w') as outfile:
        json.dump(LTP, outfile)
```

Listing 10: motorblackboard_2016.py

In the main loop, after a start-up of 100 steps, the first C-node gets activated. With this activation, the rhythm is started, too. The rhythm will take over the timing of the simulation until it is done. As it is split into groups of four, once it is done it will deactivate the C-node, too. This process than triggers the next C-node to be activated, which will in turn activate the next part of the rhythm. The simulation will stop 500 steps after the last S-node is deactivated. The 500 steps are added to wait for last activations to decline and therefore close down neatly rather then while there is still activation in the blackboard.

```
'''
Main loop
'''
if i > 100:
  activate_cnode(globc, ri)

'''
Functions
'''
"""
Activation function for main simulation run:

mb.pop_step_wc_m(i, 0, Cnode_e, Cnode_i, 20.0, 20.0):
first input (i) is time
Second input (0) is order (1e, 2e, etc) of Cnode (or Snode)
Third and fourth input (Cnode_e, Cnode_i,): the Wilson Cowan populations
Fifth and sixth input (20, 20): input for the Wilson Cowan populations
"""

# Activate C-node
# Arguments: C-node to activate, rhythm to use
# Call calculate_chance if automatic input is used and chance is not calculated yet
# cf[0] is a flag to prevent repeated calculation of the chance. Set in calculate_chance
    and deactivate_cnode
# Call rhythm function on the given arguments
# Call mb.pop_step_wc_m to activate given C-node if the simulation is still running
# r[-2][-1] is the valid value in the rhythm array. The if is added because the
    activation wouldn't stop otherwise
def activate_cnode(c, rit):
    if cf[c] == 0:
      calculate_chance(c)
    rhythm(rit, c)
    if i < r[-2][- 1] + 150:
        return mb.pop_step_wc_m(i, c, Cnode_e, Cnode_i, 20.0, 20.0)


# Deactivate C-node
# Argument: C-node to deactivate
# Use global variables globc, ri, and cf
# Global variables have to be defined because they are changed within the function but
    have to change outside, too
# Sets flag cf[globc] to 0 so chance can be calculated again in the next run
# If cf[c] == 1, the function is called for the first time
# In that case, if there are still C-nodes left in the song, change global used C-node
    and rhythm to the next one
# Call mb.pop_step_wc_m with value 0 to deactivate C-node
def deactivate_cnode(c):
    global globc
    global ri
    global cf
    if cf[c] == 1:
        cf[c] = 0
        if globc < len(LTP) - 1:
            globc += 1
            ri += 1
    return mb.pop_step_wc_m(i, c, Cnode_e, Cnode_i, 0.0, 0.0)


# Make S-node flag
snodeflag = [0] * sn
```

```
# Activate S–node with C–node
# Arguments: S–node to activate , C–node to activate it with
# Uncomment if loop to print when a S–node is active exactly once (visual feedback)
# Call mb.pop_step_wc_m to activate given S–node with the present activation of the given
     C–node
def activate_snode_with_cnode(snode, cnode):
    #if snodeflag[snode] == 0:
        #print 'S–node', snode + 1, 'active'
        #snodeflag[snode] = 1
    return mb.pop_step_wc_m(i, snode, Snode_e, Snode_i, 0.2 * Cnode_e[i, cnode], 0.2 *
    Cnode_e[
        i, cnode])


# Activate S–node
# Argument: S–node to activate
# Uncomment if loop to print when a S–node is active exactly once (visual feedback)
# Call mb.pop_step_wc_m to activate given S–node with standard activation
def activate_snode(snode):
    #if snodeflag[snode] == 0:
        #print 'S–node', snode + 1, 'active'
        #snodeflag[snode] = 1
    return mb.pop_step_wc_m(i, snode, Snode_e, Snode_i, 20.0, 20.0)


# Deactivate S–node
# Set flag back to 0
# Call amb.pop_step_wc_m with value 0 to deactivate S–node
def deactivate_snode(snode):
    if snodeflag[snode] == 1:
        snodeflag[snode] = 0
    return mb.pop_step_wc_m(i, snode, Snode_e, Snode_i, 0.0, 0.0)


# Run rhythm
# Arguments: rhythm to be used, active C–node
# r is the rhythm array
# r[rhyt] is the present active chunk of the rhythm
# r[rhyt][0] is the start time of the first S–node r[rhyt][1] its end time. resp. for
    other S–nodes
# Call activate_snode_with_cnode on the first S–node between the first two positions of
    the rhythm
# Call deactivate_snode in the break between first and second S–node
# Call activate_snode for all following S–nodes if time is a begin time (even index)
# Call deactivate_snode for all following S–nodes if time is an end time (uneven index)
# If the last step of the rhythm is reached, call deactivate_snode on last S–node,
    # and if there is a C–node left call activate_cnode on this C–node
# If last S–node time is exceeded by 150, deactivate used C–node (done later to leave
    last S–node time to deactivate)
def rhythm(rhyt, cnode):
    if r[rhyt][0] < i < r[rhyt][1]:
        activate_snode_with_cnode(0, cnode)
    if r[rhyt][1] < i < r[rhyt][2]:
        deactivate_snode(0)
    for w in xrange(len(r[rhyt]) − 1):
        if r[rhyt][w] < i < r[rhyt][w + 1] and w % 2 == 1 and w != 1:
            deactivate_snode(w / 2)
        if r[rhyt][w] < i < r[rhyt][w + 1] and w % 2 == 0 and w != 0:
            activate_snode(w / 2)
    if i > r[rhyt][- 1]:
        deactivate_snode(sn - 1)
        if cnode < len(r) - 2:
            activate_cnode(globc + 1, ri + 1)
    if i > r[rhyt][- 1] + 150:
        deactivate_cnode(globc)
```

Listing 11: motorblackboard_2016.py

Concatenation In case a learned sequence is longer than four, a following chunk has to be loaded after the first finished. The process of connecting chunk nodes with each other is described as concatenation in the literature. Implementing concatenation leads to more interaction. The program has to know which chunk node is next in order to load it. To be able to do that, lists of all chunk nodes that make up a sequence are made. These lists are stored in the C-node directory and have the names of songs. In the case of the "smoke on the water" example, the C-node is called Cnodes/smokec.txt. One could argue that C-nodes should have weights amongst each other rather than being in fixed lists. Indeed, based on the literature and logical thinking, it would be expected that C-nodes have stronger or weaker connections with each other. This would mean that there is room for mistakes in the transition between C-nodes, which is not given in the present structure. Actually, lists decrease the possibility of re-using a single chunk. A solution to that is using a blackboard architecture on those chunks, too. As well as exact iCub movements, this is out of the scope of this thesis and subject to further research.

5.4.2 LTP and gates

Simulating gates and LTP LTP is the activation between a C-node and a gate. Before understanding how LTP and therefore C-nodes can be learned, it is important to know how the connection between C-node and gate, LTP is implemented. LTP saves weights between a C-node and the gates. With three fingers and three S-nodes, there are nine gates, which can be represented as follows:

C-node 1	Finger 1	Finger 2	Finger 3
S-node 1	G_1^1	G_1^2	G_1^3
S-node 2	G_2^1	G_2^2	G_2^3
S-node 3	G_3^1	G_3^2	G_3^3

A specific row simulates all gates corresponding to a specific S-node. First, the first row gets activated, then the second row and lastly the third row. The columns do the same for fingers. The first column represents the gates belonging to the first finger. In matrix form, LTP of one C-node with all gates will be represented like this:

$[[G_1^1, G_1^2, G_1^3]$
$[G_2^1, G_2^2, G_2^3]$
$[G_3^1, G_3^2, G_3^3]]$

LTP in the program is a list of matrices with weights at every position. With two unlearned C-nodes, it will the representation will look like this:

$[[[0.05, 0.05, 0.05]$

[0.05, 0.05, 0.05]
[0.05, 0.05, 0.05]]

[[0.05, 0.05, 0.05]
[0.05, 0.05, 0.05]
[0.05, 0.05, 0.05]]]

After one learning cycle with three S-nodes, three of the weights will be updated. What weights is depending on the active C-node, S-node, and feedback. Assume that while the first S-node was active, feedback was given from finger two. While the activation of the second S-node, feedback was given from finger three. In the time the third and last S-node was active, feedback came from finger one. In this situation, LTP will be updated like that:

[[0.05, 0.0505, 0.05]
[0.05, 0.05, 0.0505]
[0.0505, 0.05, 0.05]]

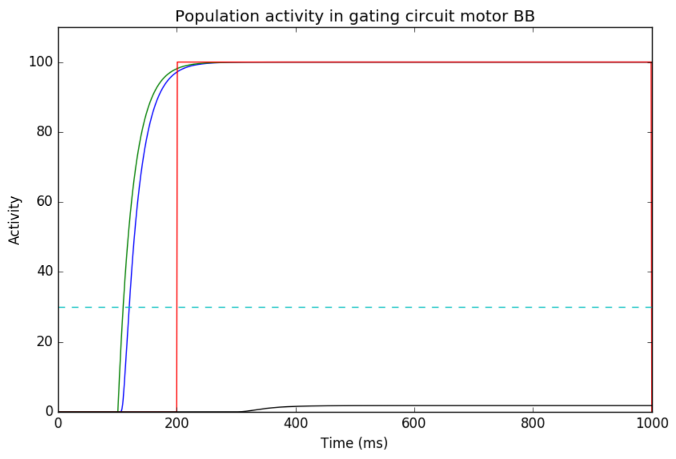
The bigger a weight, the stronger the connection between gate and C-node. The weight values are discussed in the next section. The manner in which LTP learning is implemented is shown in listing 12.

```
# Check if learning condition is given
# Search for active C-node
# If C-node is active, search for active S-node
# If S-node is active, search for gates in which:
    # feedback is given (pop6 active),
    # pop3 activation is about the learning threshold,
    # and no flag has been set yet
# If such a gate exists, call learn function with found C-node, S-node and gate as
    arguments
def activate_learning():
    for cnode in xrange(cn):
        if Cnode_e[i, cnode] > 99:
            for snode in xrange(sn):
                if Snode_e[i, snode] > 95:
                    for finger in xrange(gate):
                        if p6_e[i, finger] > 95 and p3_e[i, finger] > learning_threshold
                                and LTP_flags[cnode][snode][finger % f] == 0:
                                    learn(cnode, snode, finger % f)

# Update LTP (learn)
# Arguments: Active C-node, active S-node, active finger
# Set LTP to present value times the learning_factor
# LTP[cnode] is a matrix with as much positions as gates
# LTP[cnode][pos][finger] is the weight between the C-node and the gate (S-node and
    finger define the gate)
# list_cnodes is a list of which finger has to be played in which position based on the C
    -node
# Print correctness of learning
# Set flag of gate to 1 so learning is only done once per run
def learn(cnode, pos, finger):
    LTP[cnode][pos][finger] *= learning_factor
    if finger == list_cnodes[cnode][pos]:
        print "I learned right!"
    else:
        print "I learned wrong :("
    LTP_flags[cnode][pos][finger] = 1
```

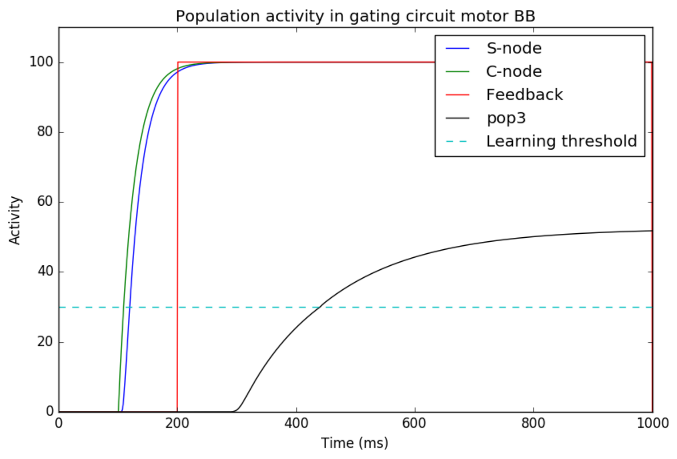
Listing 12: motorblackboard_2016.py

The only thing that has to be done in the simulation run itself is to call the *activate_learning()* function, which will trigger learning based on the C-node, S-node, and feedback.

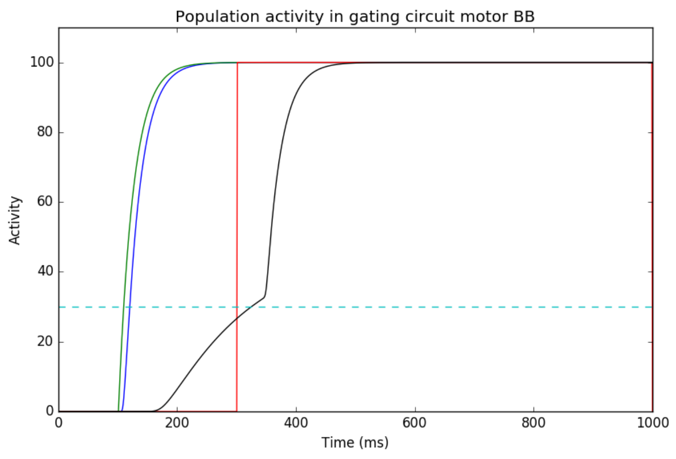
Weights LTP of a C-nodes is learned if pop3 of a specific gate and the C-node are active at the same time. In order of learning to be facilitated, there has to be a strong enough relation between them in the beginning. Figure 16 shows how pop3 responds given different weights. A weight can have a value between 0 and 1. It will determine how much of the activation of the C-node will affect pop3.



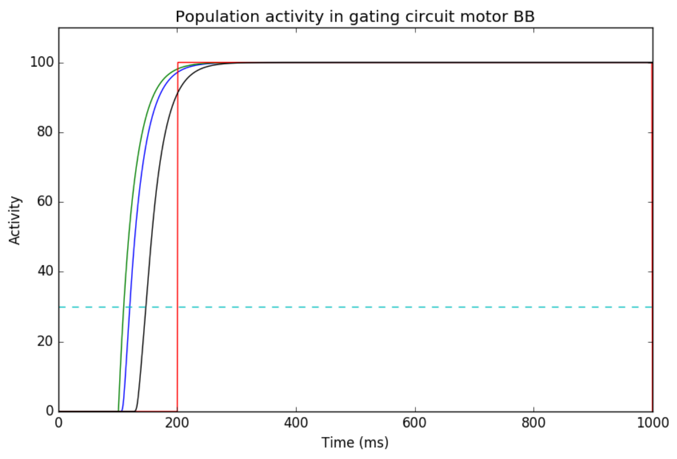
(a) Weight: 0.03



(b) Weight 0.05



(c) Weight 0.25



(d) Weight 0.35

Figure 16: Effect of LTP on pop3 activation

As illustrated in the graphs, a weight of 0.03 is not enough to reach the learning threshold of 30, even with endless amounts of time. To reach the learning threshold, a weight higher than 0.045 is needed. With a weight of 0.05 activation can reach an activation of 50 given enough time. This is why 0.05 was chosen as the begin weight for all simulations. With low weight values, the simulation can be said to be in reaction mode. A question that arises is how the weight of

0.05 is achieved. It can be argued that this activation is established while learning how to move fingers. Another explanation may be that the learning threshold is lower when beginning to learn. The problem with this idea is that if a learning threshold is too small, everything will get learned very quick, even if that is not desired.

Figures 16c and 16d show how pop3 behaves when weights get higher. A weight of 0.25 will be enough to trigger the finger to move without feedback, but if feedback is given its speeds up activation. Given such a weight, the program simulates the associative mode, in which cognitive feedback is needed for fast sequence productions. With a weight of 0.35, activation of pop3 is high enough to trigger a quick response based on C- and S-node activation alone. In this stage, the program is modeling the chunking mode.

5.4.3 Fingers and Feedback

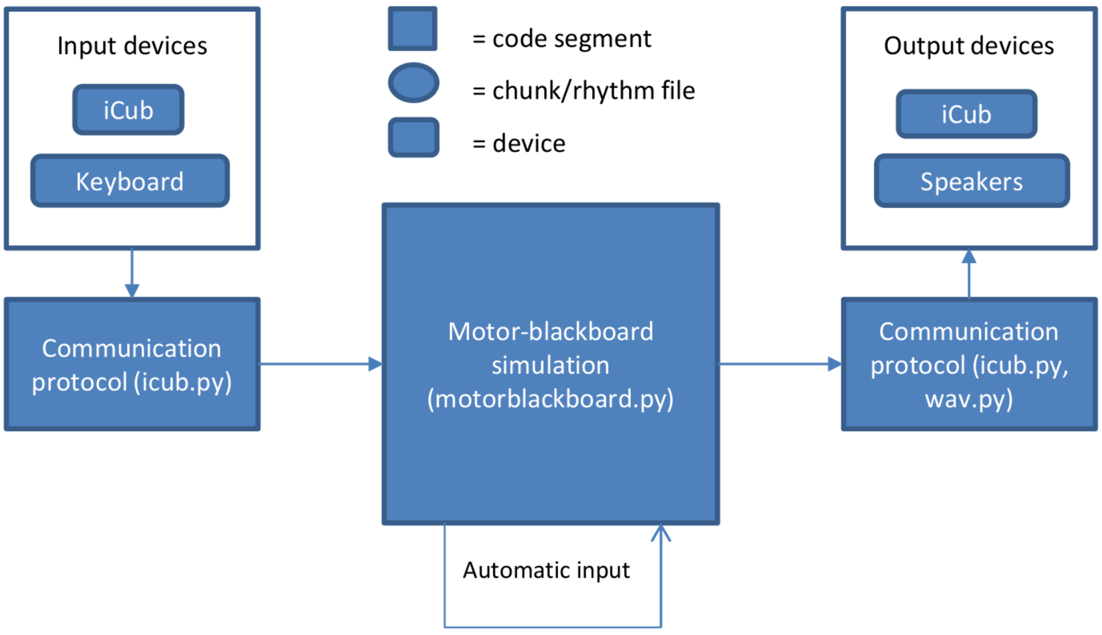

Input devices
iCub
Keyboard
= code segment
= chunk/rhythm file
= device
Output devices
iCub
Speakers
Communication protocol (icub.py)
Motor-blackboard simulation (motorblackboard.py)
Communication protocol (icub.py, wav.py)
Automatic input

Figure 17: Components used for feedback and for finger activation

Feedback is given at the moment a finger is moved. Without finger movement, no learning can take place. There are different ways of adding feedback to the simulation, with or without the iCub. The three possibilities in the program are: “smart” automatic feedback, feedback through keyboard input, and feedback from the iCub.

When implementing any feedback, it is important to consider the S-nodes. This is because feedback does not have any use without a corresponding gate being activated. In figure 18 S-node activation, pop3 activation and feedback for one gate are shown in one color. For both S-nodes, there is feedback activation, but pop3 is only activated in the first step, not in the second. This is because feedback of the first step is aligned, while the feedback is too late to trigger a gate reaction in the second step. It could also be the case that late feedback interferes with further learning, as illustrated in figure 19.

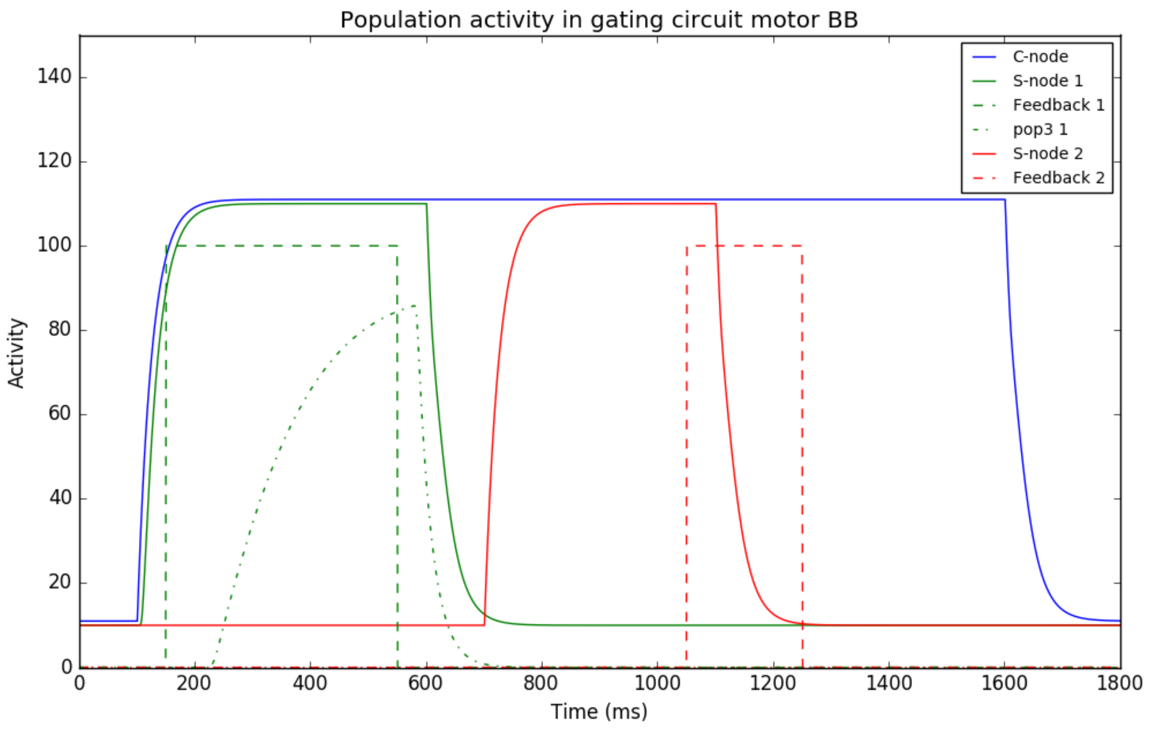


Figure 18: Effect of normal and too little feedback on pop3 activation.

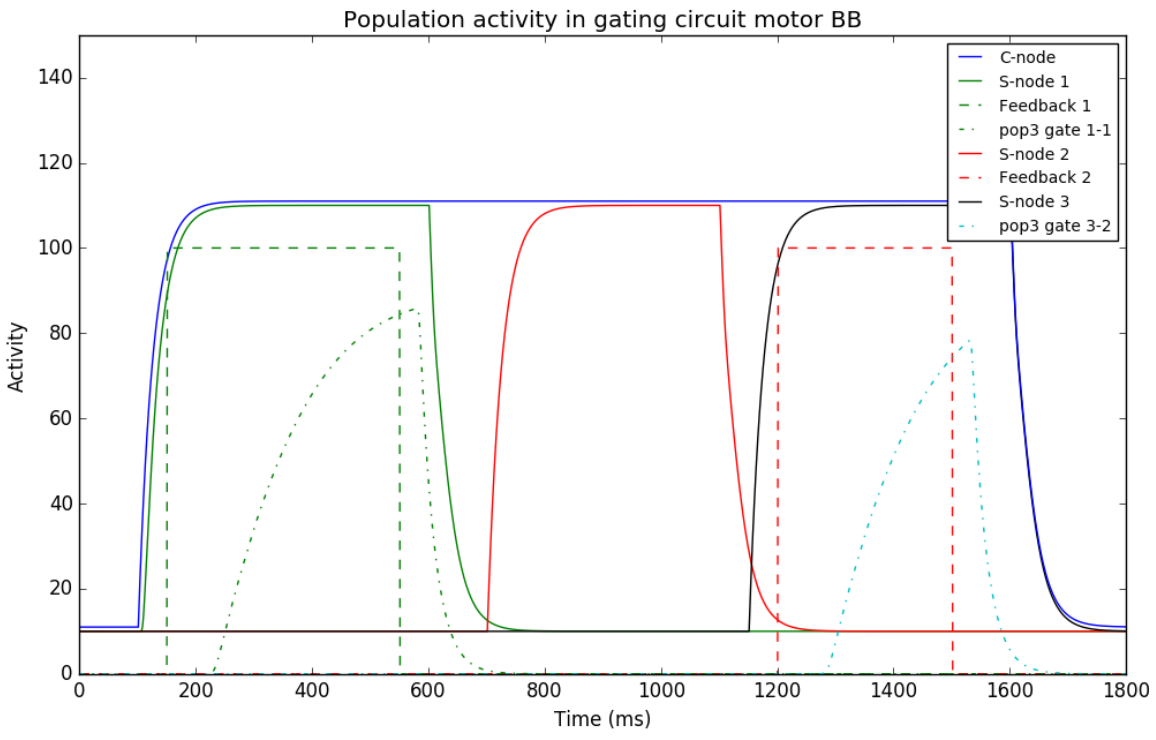


Figure 19: Effect of too late feedback on pop3 activation in several gates.

Due to late feedback in the situation showed in figure 19, wrong associations are learned. The gates that should have been learned are G_2^2 and G_3^3. Instead of these two gates, the connection between C-node and gate G_3^2 was strengthened. This is the gate corresponding to the second finger and the third S-node.

Another interesting observation can be made. The activation of pop3 in the first learning event is bigger than the activation of the pop3 in the second learning event. This is due to shorter feedback, and less time in which pop3 is activated. This phenomenon occurs when weights are not high, yet. As discussed in the previous section, low weights lead to less quick activations of pop3, which is an indicator that learning in earlier stages will take longer, and movements will be slower as a result. This section is about how the problem of synchronizing S-nodes and feedback in early learning stages was dealt with during the implementation. Listing 13 shows the general activation of feedback.

```
'''
Main simulation
'''
# choose which input version is used:
if use == 0 or use == 3:
  feedback_automatic(ran1, ran2, ran3)
if use == 1:
  feedback_key()
if use == 2:
  feedback_iCub(30, 10)


'''
Feedback activation to be called by the functions
'''
# Make feedback flag
fbackflag = [0] * f


# Activate Feedback
# Argument: Finger that gives feedback
# Set feedback of the given finger (i = present step of the simulation) to 100 (max)
# Uncomment if loop to print when feedback is active exactly once (visual feedback)
# Return feedback
def feedback_on(finger):
    Fback[i][finger] = 100
    #if fbackflag[finger] == 0:
        #print 'Feedback from finger:', finger + 1
        #fbackflag[finger] = 1
    return Fback[i][finger]


# Deactivate Feedback
# Argument: Finger of which feedback has to be turned off
# Set feedback of the given finger (i = present step of the simulation) to 0
# Set flag back to 0
# Return feedback
def feedback_off(finger):
    Fback[i][finger] = 0
    if fbackflag[finger] == 1:
        fbackflag[finger] = 0
    return Fback[i][finger]
```

Listing 13: motorblackboard_2016.py

Automatic feedback Automated feedback simulates the input that would normally come from cognition. As discussed above, feedback has to come at the right moment, namely when the correct S-node is active. When automating the process that is little of a problem, because we can make feedback dependent on S-nodes. This is done by activating it a bit later than the S-node and stopping a bit earlier than the S-node. Consider the following code of activating feedback for finger one, while S-node one is active.

```
# Manage automatic feedback, does at the moment only work with three fingers
# Arguments: random numbers for every finger
# Feedback is activated 50 steps after the S-node and deactivated 50 steps before the
    deactivation of the S-node
# r[ri][0] is the first position of the active rhythm
# chance[0][1] is the chance that finger two is activated at the first position. In the
    beginning the chance is 15%
# If none of the "wrong" choices is valid, the "right" feedback is activated
def feedback_automatic(ran1, ran2, ran3):
    if i > r[ri][0] + 50:
        if ran1 < chance[0][1]:
            feedback_on(1)
        elif ran1 > 1.0 - chance[0][2]:
            feedback_on(2)
        else:
            feedback_on(0)

    if i > r[ri][1] - 50:
        if ran1 < chance[0][1]:
            feedback_off(1)
        elif ran1 > 1.0 - chance[0][2]:
            feedback_off(2)
        else:
            feedback_off(0)

  # add for other fingers respectively
```

Listing 14: motorblackboard_2016.py

There is more to this feedback than just turning on the feedback of the good finger. The function *feedback_on()* turns on the feedback for a specified finger, but there is also a chance that another finger will be giving feedback. Chance is added to simulate mistakes, in order to make learning more realistic. To simulate mistakes, *random()* is used and compared to the chances. Random is calculated per learning loop to make the begin chance of mistakes normally distributed over time.

Take the code example in listing 13. The correct finger to move would be finger 0. If random is less than 0.15, not finger 0, but finger 1 will be activated. Similar, for a random above 0.85, finger 2 will be activated. In the example, the program has a 70% chance to learn right in the beginning. The more the program has learned, the more chance will be allocated to the learned finger. The following part of code shows how that is dealt with.

```python
# Calculate chance for automatic learning
# Argument: C-node to calculate the chance for
# Use global variable cf
# Set flag on cf
# bcg is the begin chance of the good finger to be used, set in the beginning of the
    program
# bcf is the begin chance of a wrong finger to be learned, calculated from bcg and the
    amount of fingers
# list_cnodes is a list of which finger has to be played in which position based on the C
    -node
# LTP[cnode][pos] are all weights a C-node has at one position of the sequence
# LTP[cnode][pi][xf] is the weight between the C-node and the gate (S-node and finger
    define the gate)
# chance[pi][xf] is the chance of a finger (xf) to be activated in a position (pi)
# For every position in the C-node:
    # look up the good finger,
    # calculate the sum of all weights
    # make temporary array for chance per finger
    # for every finger:
        # calculate the weighted weight of that finger
        # calculate difference between weighted weight and expected weight (can also be
    negative)
        # add/substract difference from begin chances to correct for learned associations
     and store it in the chance array
def calculate_chance(cnode):
    global cf
    cf[cnode] = 1
    for pi in xrange(sn):
        good = list_cnodes[cnode][pi]
        sumw = sum(LTP[cnode][pi])
        c = [0]*f
        for xf in xrange(f):
            c[xf] = (LTP[cnode][pi][xf])/sumw
            c[xf] = c[xf] - 1.0/f
            chance[pi][xf] = (bcg if xf == good else bcf)+c[xf]
```

Listing 15: motorblackboard_2016.py

Though this is a fairly realistic learning procedure when applying a feasible chance of good learning, there is one point that is not yet taken to account. Take the example of a child playing piano with two hands. If it has to use the ring finger of the left hand, it is more likely that he will mistake that finger for another finger of the left hand, rather than a finger of the right hand. Also, closeness of fingers may account for the chance of making mistakes.

Keyboard input To make feedback more flexible, input devices are used. As the iCub was not available and the simulator is prone to errors and difficult to handle, the keyboard variant was introduced for prototyping. To capture keyboard actions, the Pygame module was used. During the implementation of the prototype, it became clear that it is really hard to hit the keys at the exact time, even when making the steps of the simulation slower. While thinking about making it easier the idea of a training rhythm came to mind. Thinking about a child playing piano. It most likely will begin with pressing the keys slow and without any rhythm. So for the early stages a rhythm that does exactly that - being slow and monotonous - should be used.

```python
# Manage feedback from keyboard
# Assign keys to fingers and call feedback_on if the key is pressed
# Get all pressed keys through pygame event
# If a key is pressed, call feedback_on on respective finger
# If a key is released, call feedback_off on respective finger
def feedback_key():
    pygame.event.pump()
    pressed = pygame.key.get_pressed()

    if pressed[pygame.K_1] == 1:
        feedback_on(0)
    else:
        feedback_off(0)

    # add for other fingers respectively
```

Listing 16: motorblackboard_2016.py

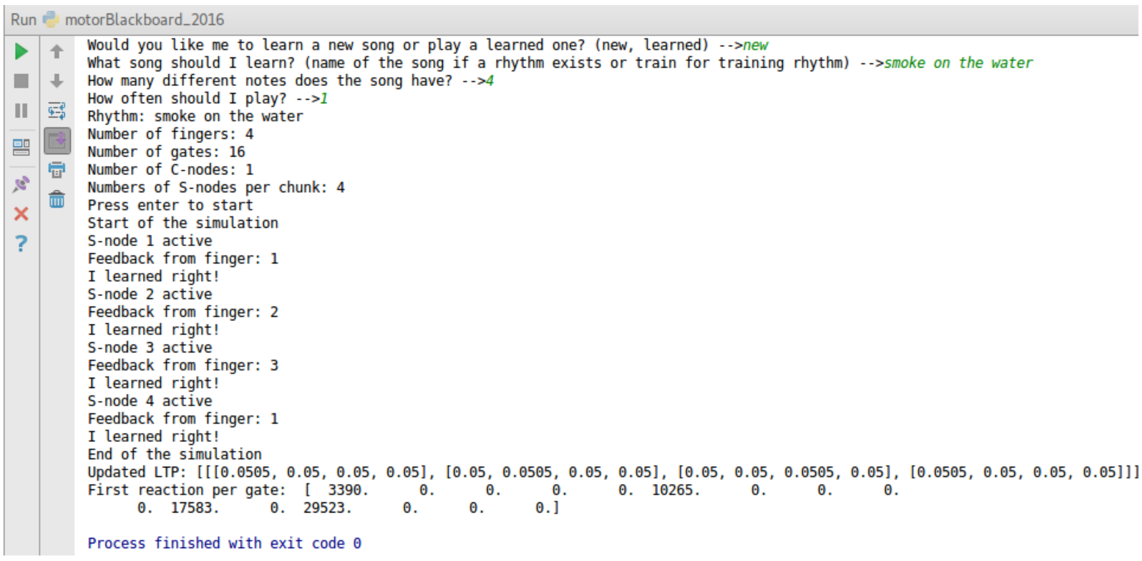
iCub The last step is implementing learning with the iCub. It is chosen to pin an angle of a finger motor to the activation of feedback. Angles for the iCub fingers have a value of zero when they are stretched out. The angle is used as input to the iCub learning simulation. If the angle is above a certain value given in the simulation, feedback is activated. If it decreases to an angle smaller than the minimum activation value stated, feedback is turned off. The *icub.get_Pos()* function is discussed in the rhythm simulation. Refer to listing 5 for the code.

```
# Define how iCub data should be interpreted
# Arguments: minimun and maximum angle
# call function icub.get_Pos to receive angle parameter
# If angle is bigger than the max value, turn feedback on that finger on
# if finger is returned under the minimal value, turn feedback off
def feedback_iCub(maxi, mini):
    if icub.get_Pos(0) > maxi:
        feedback_on(0)

    if icub.get_Pos(0) < mini:
        feedback_off(0)

    # add for other fingers respectively
```

Listing 17: motorblackboard_2016.py

Figure 20 shows how a C-node is learned for the first time. The speed in this trial was set to 15, which is around ten times slower than the simulation can play once a sequence is learned. This accounts for the high reaction time values. Slowing the simulation down is necessary to capture the right times for learning as a user. The fact that this results in high reaction times is conform to the theory that learning is slow in the beginning. The fact that it is this high probably stems from the fact that hitting keys is based on visual activation of the S-nodes. If the user of the program is not used to that, a slow simulation assures right learning.

Finger activation after learning The motor-blackboard theory states that a finger gets activated when pop2 of a corresponding gate is active. At some point, that will happen without automatic or manual feedback. In that case, the program automatically activates the appropriate feedback and output based on the devices chosen as output. Refer to listing 18, 19 and 20 for the code.



```
Would you like me to learn a new song or play a learned one? (new, learned) -->new
What song should I learn? (name of the song if a rhythm exists or train for training rhythm) -->smoke on the water
How many different notes does the song have? -->4
How often should I play? -->1
Rhythm: smoke on the water
Number of fingers: 4
Number of gates: 16
Number of C-nodes: 1
Numbers of S-nodes per chunk: 4
Press enter to start
Start of the simulation
S-node 1 active
Feedback from finger: 1
I learned right!
S-node 2 active
Feedback from finger: 2
I learned right!
S-node 3 active
Feedback from finger: 3
I learned right!
S-node 4 active
Feedback from finger: 1
I learned right!
End of the simulation
Updated LTP: [[[0.0505, 0.05, 0.05, 0.05], [0.05, 0.0505, 0.05, 0.05], [0.05, 0.05, 0.0505, 0.05], [0.0505, 0.05, 0.05, 0.05]]]
First reaction per gate:  [  3390.      0.      0.      0.      0.  10265.      0.      0.      0.
      0.  17583.      0.  29523.      0.      0.      0.]

Process finished with exit code 0
```

Figure 20: Training the first C-node of smoke on the water. LTP weights begin at 0.05. Weights of learned gates get updated to 0.0505 with a learning factor of 1.01. Reaction time is measuring how long population six of a gate took to gain an activation of 90.

```
# Make gate activation flag
finger_flag = [0] * gate


# Manage the output of the simulation
# If there is a gate with a pop2 activation above 90, calculate the corresponding finger
    # Call feedback_on on that finger
    # If there was no output action on this finger yet (fingerflag[fingers] == 0):
        # if the iCub is used, call icub_movefinger
        # call wav.play_note with the finger and the song as argument
        # Set flag to 1
# If a finger has less than an activation of 10
    # turn feedback off
    # if the iCub is used, call icub_returnfinger
    # set flag back to 0
def manage_output():
    for fingers in xrange(gate):
      finger = fingers % f
        if p2_e[i, fingers] > 90.00:
            feedback_on(finger)
            if finger_flag[fingers] == 0:
                if use == 2 or use == 3:
                    icub.movefinger(finger)
                print "Finger activated: ", finger + 1
                wav.play_note(finger, songs)
                finger_flag[fingers] = 1
        if 90 > p2_e[i, fingers] > 0.1:
            feedback_off(finger)
            if use == 2 or use == 3:
                if finger_flag[fingers] == 1:
                    icub.returnfinger(finger)
            finger_flag[fingers] = 0
```

Listing 18: motorblackboard_2016.py.py

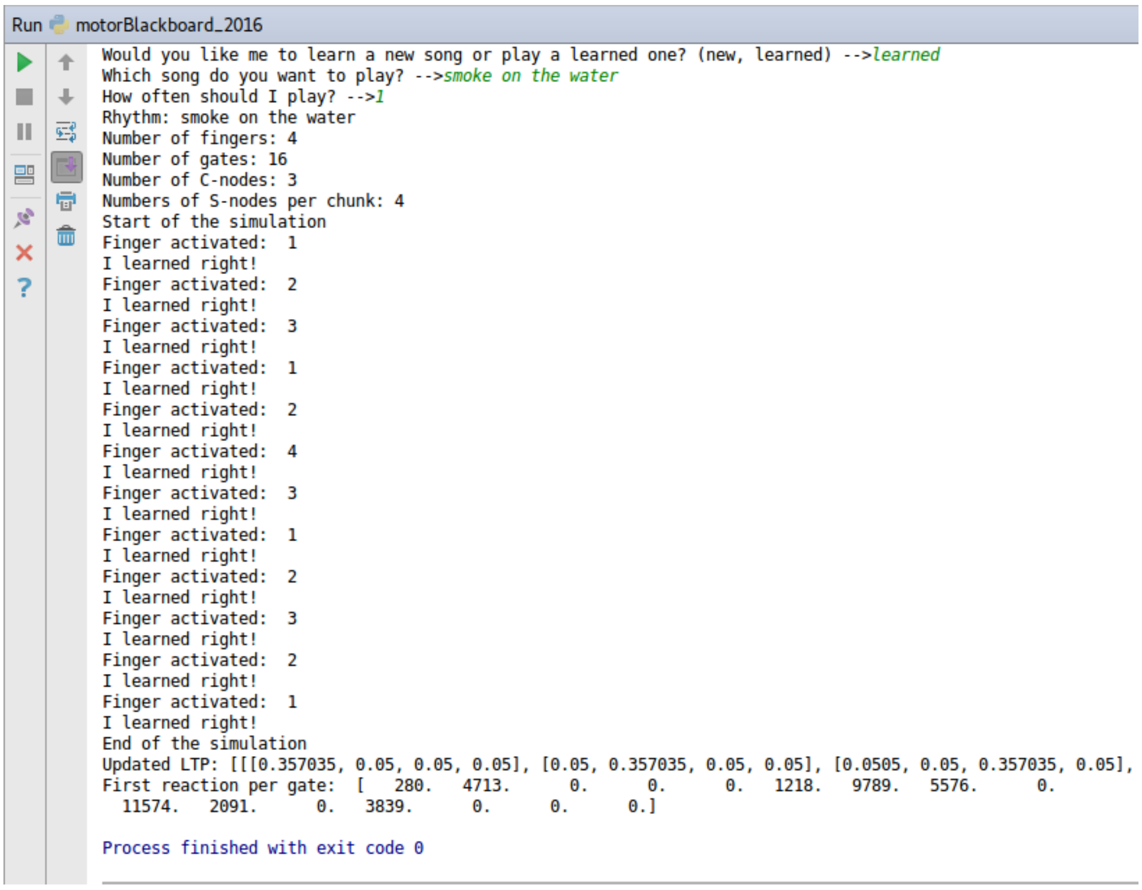
```python
# Move a finger of the iCub
# Arguments: number of a finger
# If a finger has to be moved, set temporary array to new value and move finger to that
    position
def movefinger(finger):
    if finger == 0:
        tmp.set(10, 80)
        iPos.positionMove(tmp.data())

    # add for other fingers respectively


# Return a finger of the iCub
# Arguments: number of a finger
# If a finger has to be returned, set temporary array to standard value and move finger
    to home position
def returnfinger(finger):
    if finger == 0:
        tmp.set(10, 10)
        iPos.positionMove(home.data())

    # add for other fingers respectively
```

Listing 19: icub.py

```python
# Map fingers to notes
# First if statements define the song
# Second if statements set sound per finger
def play_note(finger, song):
    if song == 'smoke':
        if finger == 0:
            sound = pygame.mixer.Sound("g3.wav")
            sound.play()

        # add for other fingers respectively
```

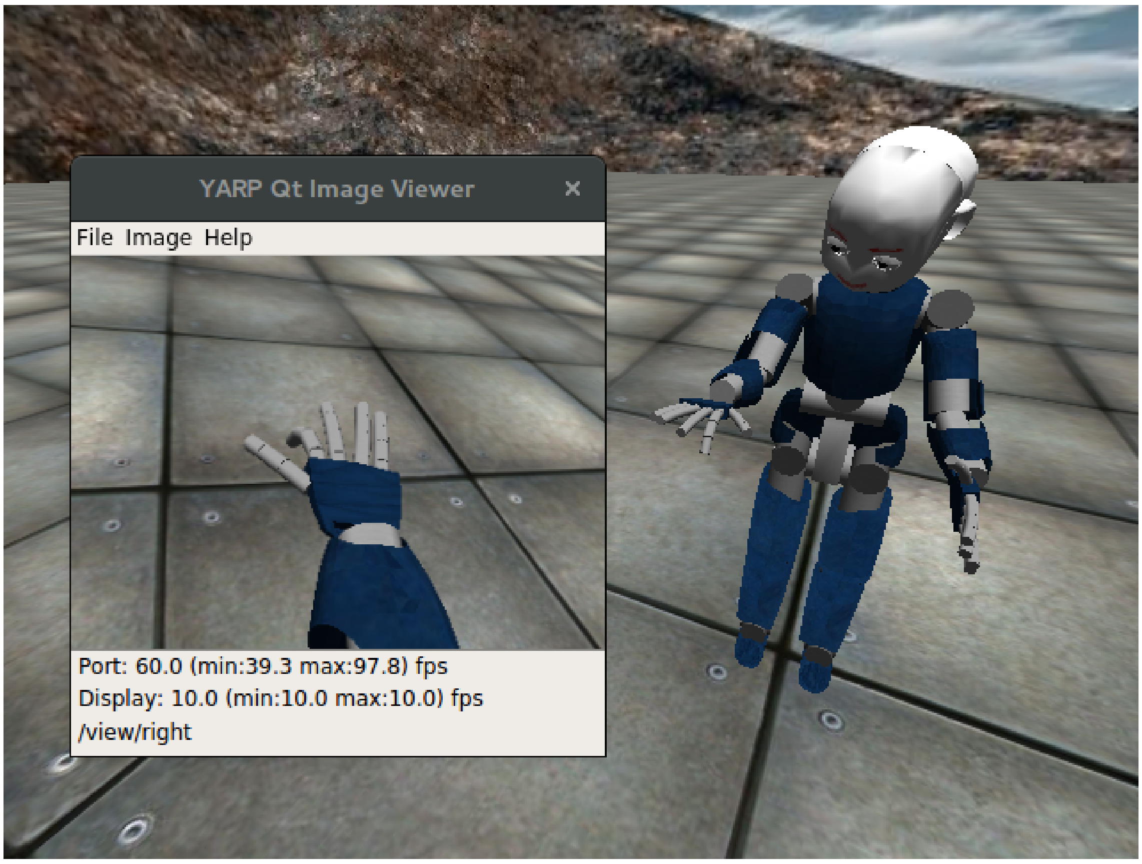
Listing 20: wav.py

Once the simulation is learned once it can be run in the “learned” mode. Based on the learning factor, it will take numerous trials to reach an activation that triggers a fully automatic reply. How the program behaves in this situation can be seen in figure 21. Note that the third position was not learned perfectly. There was a point in learning, where finger one instead of finger three was used in this position. Also, reaction time is much quicker than in figure 20, due to the speed setting that can now have a lower value. Figure 22 shows how the iCub would behave when set as output device to the learned sequence.



```
Would you like me to learn a new song or play a learned one? (new, learned) -->learned
Which song do you want to play? -->smoke on the water
How often should I play? -->1
Rhythm: smoke on the water
Number of fingers: 4
Number of gates: 16
Number of C-nodes: 3
Numbers of S-nodes per chunk: 4
Start of the simulation
Finger activated:  1
I learned right!
Finger activated:  2
I learned right!
Finger activated:  3
I learned right!
Finger activated:  1
I learned right!
Finger activated:  2
I learned right!
Finger activated:  4
I learned right!
Finger activated:  3
I learned right!
Finger activated:  1
I learned right!
Finger activated:  2
I learned right!
Finger activated:  3
I learned right!
Finger activated:  2
I learned right!
Finger activated:  1
I learned right!
End of the simulation
Updated LTP: [[[0.357035, 0.05, 0.05, 0.05], [0.05, 0.357035, 0.05, 0.05], [0.0505, 0.05, 0.357035, 0.05],
First reaction per gate:  [   280.    4713.      0.      0.      0.    1218.    9789.    5576.      0.
  11574.    2091.      0.    3839.      0.      0.      0.]

Process finished with exit code 0
```

Figure 21: Playing smoke on the water after intensive training. The speed setting in this run was set to two. This accounts for the short reaction times.



Port: 60.0 (min:39.3 max:97.8) fps
Display: 10.0 (min:10.0 max:10.0) fps
/view/right

Figure 22: The iCub simulator looking at his hand while playing the second finger. The embedded picture is the camera view of the right eye.

6 Discussion

6.1 Revisiting the motor-blackboard

In the previous section it was shown that the iCub can in fact learn to play piano using a motor-blackboard architecture. This section will discuss the findings, what they indicate and where further research is needed.

6.1.1 Handling gates and rhythm

Rewind to the architecture of the motor-blackboard. The number of gates per finger is dependent on the amount of S-nodes in a sequence. But how does the brain know how many gates it should have per finger/action to perform? Isn't it more likely that there is a fixed number of steps in a rhythm chunk, too? And that more of these chunks form melodies? The fact that children learn to recognize rhythm in phrases, before recognizing melodies, speaks for the fact that rhythm can be chunked. Also, it would make gates easier to manage, as there would only have to exist around four for every finger/action. Of course, that leaves us with the question of how long these chunks should be.

The easiest way to implement the feature of playing more chunks after each other was to make the rhythm chunks as long as motor chunks. In that way the first initialization sets a song, which refers to one or more chunks and a rhythm, which has to contain at least the same amount of rhythm chunks. This is because learning a sequence without rhythm is impossible; the S-nodes have to activate the gates.

Sakai, Hikosaka, and Nakamura (2004) found that, when no rhythm is given with a sequence to be learned, that people tend to develop their own rhythms in a 1:2, or 1:3, ratio. This might be a result of chunking or vice versa. Chunking is not a new concept and it is fairly certain that it is correct. But there is less knowledge about the connection with rhythm. Based on (computational) logic, it is proposed that the execution of rhythms is subject to changes while running. This is opposed to C-nodes which would only change behavior between runs. Rhythms can be seen as patterns and patterns are the basis of all moving life. The more powerful a creature is in manipulating and using patterns, the more successful it will be. So while C-nodes are solid knowledge, rhythms might have to be monitored in order to adapt to surroundings. Another characteristic is that rhythms can be used in every blackboard situation, no matter how high or low the level. Based on this consideration, it is proposed that rhythm and knowledge are two fundamentally different concepts. While lists of C-nodes, as well as movements could be learned in a motor-blackboard manner, this is not the case for rhythms. More research should be done on how rhythms are learned and how they influence our lives.

6.1.2 C-nodes and Concatenation

As mentioned earlier, there has to be some container for the chunk nodes. Throughout the implementation they were referred to as names of songs. Here it is proposed to call the action nodes. Action nodes make possible to play a sequence that is longer than four, but could also be used to make a movement which has more than four reference points. In the simulation, a user calls an action node by stating the name of the song. This will, in a separate process, also trigger the rhythm of this song.

The introduction of action nodes is the implementation of the theory of initiation and concatenation. In the beginning of playing a sequence, the song is loaded and the first C-node starts. Once the first C-node is done, other C-nodes can follow with less initiation time as the song is already loaded into the simulation or in other words, the blackboard.

6.1.3 Fingers

In theory, population 2 of each gate activated a so called "finger". This might be true for the discrete sequence production task where fingers are pressing the same key throughout the task. But as a piano has way more than ten positions, the positions of the hands also have to be accounted for. So actually in this case the expression shouldn't fingers but action, as the gate will trigger more than one joint movement in a finger. In the implementation this is accounted for through the translation files icub.py and wav.py, depending on the output mode. They can be seen as translations between actual movements and the sequence production in the brain. In a further step, these translations could also be expressed in C-nodes and rhythms, which would make them easier to manage. Also, code could better be reused with such a structure, as no chunk would saved twice.

6.1.4 Feedback

First of all, it has to be noted that feedback was not treated as a population that can be activated but rather as a component that immediately has the same impact as a fully activated population. Future work should research how feedback is given. Although fingers could be treated as a population, there might be different concepts that fit even better.

The way of giving and receiving feedback in a realistic manner was the most complex part to implement. This is not surprising, as feedback is the crucial part in learning. Feedback at the wrong time will either result in wrong learning or no learning at all. Though wrong learning might happen at the moment a mistake is made, after that not learning is more realistic. This is because learning in the beginning is highly dependent on cognitive processes. In the piano example that means that when hearing a wrong note, or if the rhythm doesn't seem to be right anymore, the learning trial will be stopped. At the moment, the automatic learning part of the program does not support that. The result of that is that the program and therefore the iCub can learn more than one mistake per trial.

When learning music, mistakes can be easily recognized. This is because mistakes can be heard immediately. When using an input device, the person using the program can fall back to his or her own cognition. For the automatic program to know when a mistake is made, more input is needed. It would have to know every C-node that is to be learned. Though this is possible, the added value for this thesis was not big enough and it was chosen to be more flexible with the length of sequences, rather than making automated learning more sophisticated. After all, when learning with the iCub or a keyboard, the executing person can stop a trial if something goes wrong.

The problem of adding feedback at the good point in time, which is no piece of cake, even when done by a person, speaks for the idea that there might be a learning rhythm that is slow and monotonous. Such a rhythm would facilitate early learning through reducing cognitive workload. Another option to deal with this problem could be the implementation of an adaptive rhythm. At the moment after the start of a rhythm there is no control over it. Trying to catch up with such a rhythm can be hard. It is proposed that a rhythm is either adapted to the time the learner takes for the first step, or that there is a cognitive process that monitors “readiness” of the learner to go on. Such an adaptive rhythm would save time in such a sense that the learner does not have to start over every time he misses a step.

Another component that is rather straight forward to improve in automated feedback is the chance calculation for mistakes. In the present simulation, all mistakes have the same chance to be made. Of course this is not how it works with real mistakes. The chance of a mistake with a finger close to the right one is higher than the chance of a finger that is further away. Also, it is less common to make a mistake with the other hand, than interchanging fingers on one hand. This is due to closeness of motor control in the primary motor cortex (Kalat, 2009). To define a good mistake calculation, mistakes should be studied and the statistics should be translated to the program. A problem with this is that with every tasks, the numbers might be different, due to body structure and instruments used. In conclusion, calculating chance is a big issue when trying to do it realistic. It is influenced by a lot of factors, some of them not even known yet.

6.1.5 Weights and learning threshold

It was seen that a begin weight resulting in an activation under 4,5% of the total activation, cannot facilitate learning. A question arising from that is how the already existing LTP is learned. An explanation might be found in the translation files. As the actual movements could also be built from an motor-blackboard architecture, they can be represented as action nodes that can be called. So when these are learned they will also have some kind of reference, which could be called by every other process. The existing activation of the LTP therefore could be a sign that every connection in the brain is possible, and that learning is strengthening desired connections. This complies with the theory if Hebbian learning, where connections are made when two neurons fire at the same time.

The learning threshold was kept at an activation of 30% throughout the process. It can be argued that this is too low, or too high. There is no clear guideline in the literature and 30% was chosen because it seemed to work fine. However, based on the activations throughout the learning process it is proposed that the learning threshold is a function of the amount of long term potential between a C-node and a gate. As with most computational task, the logarithm might be the best fitting function to describe the threshold.

In the present simulation, weights get higher through learning, but are never lowered again. This is not representative for reality, where associations that are not used are weakened after a while. (Gleitman, Gross, & Reisberg, 2011). Of course, a simulation would have to run for a longer time to make this a useful feature. And even if it would be implemented, another problem arises, as previous learned behavior is learned quicker when re-learned, or even with a sudden revival (Gleitman et al., 2011). Although the learning factor can be increased in such a scenario, the logical problem is how the knowledge that something was once learned is gained. An option would be to add a field to the C-node files that indicate what the highest LTP learned was. Such a field could identify re-learning sequences so that the program could give them a higher learning factor.

6.2 Dual processor

Both processors, cognitive and motor, can be found back in the program. Everything that needs input from outside, which is essentially feedback, would in the early staged be done through cognitive processes. Everything automatically given through the motor-blackboard-dynamics is representing the motor process. As in the literature, the motor processor gets more important and the cognitive processor less important, based on the programs level of sophistication on a sequence.

The fact that the program focuses on motor processing strengthened the idea of the two processors. The later the program is in learning, the easier it is running on its own, without complicated feedback functions. As discussed above, simulating cognition (feedback) with the iCub, or the keyboard is less work than with a program. This is of course because the cognition of the brain of the operator can be used, instead of an implementation of those processes.

Abrahamse et al., 2012 discuss that initiation and execution of sequences are two different processes, meaning that playing a melody with another finger combination will take longer to execute while the initialization time will stay the same. At first, the motor-blackboard does not seem to support such a mechanism, as gates are fixed to one finger and therefore another execution will be totally different and without learned LTP. But actually letting the iCub play a learned sequence with other fingers will only need a bit of computation, namely a change in the translation protocol between the motor-blackboard and the iCub. This computation can be seen as modeling the cognitive process of for example switching finger one of the right hand to finger one of the left hand. Of course, such a protocol could be made in a way that factors like biometric easiness of translation are taken into account.

6.3 Artificial neural networks

The motor-blackboard is a solid neurally inspired neural network. It leads to a logical implementation of learning that can be extended gradually. The different components do not only help the models resilience in terms of changes, but also invite new components to be added. Furthermore, it is predestinated to be used on neuromorphic hardware, which could enhance performance even more than rewriting parts in C.

From a computational point of view modules are handy to work with. The advantage is being able to look at parts as black boxes where only in and output matter. Through adding new, and improving existing parts, such an architecture can grow into a sophisticated artificial neural network. Through new technical developments, take quantum computing as an example, concepts can get more complex and new insights about humans might be gained as well.

7 Conclusion

The goal of this thesis was to implement the motor-blackboard theory on the iCub, using the example of playing the piano. Though there are still some open questions, it was shown that the framework can in fact make the iCub learn in a child-like manner. While implementing the theory, some concepts were strengthened while others raised questions.

In conclusion, learning the iCub to play piano should be done in five phases. First, a rhythm should be learned. Then, the specific movements used to play the piano have to be acquired. After that, sequence learning will start. The program will go through the reaction mode, where it heavily relies on feedback. Following some practice, it will be in associative mode, only relying on feedback if a sequence has to be played quick. Finally, the program will evolve to chunking mode, where only initialization will be needed to play a sequence without further input. The fact that the three phases of sequence learning arose from the implementation strengthens the motor-blackboard theory.

8 Further research on the iCub

Unfortunately, the iCub code could only be tested on the virtual iCub simulator and not the real robot. Though these two are the same regarding the programming, problems that only arise when using the actual robot might not be considered well enough. Also, movements are kept very simple and are therefore not totally representative. This is not a problem here, as the motor learning stays the same, the difference lies in how the iCub interprets the signals he gets. In order to let the iCub play for real, the coding of the translation file should be refined. Think for example of several joints to manipulate per key that has to be pressed. Not only the finger should be considered, but also the position of the arm. As discussed before, this could also be done with a motor-blackboard architecture. If all positions are coded, either through hard-coding or another learning program, the only thing left to do is give numbers to each key, learn the sequence, and watch the iCub play.

The next big step of the project would be adding cognition to the model to be used on the iCub. That will require the use of perception, both visual and audible as well as knowledge and sense-making about the given inputs. When establishing further components, the most important thing to keep in mind is the communication between code fragments. As long as this is done in the same format, later changes to any part of the code should not affect the rest of the program.

References

Abrahamse, E. L., Ruitenberg, M. F. L., de Kleine, E., & Verwey, W. B. (2012). Control of automated behaviour: Insights from the Discrete Sequence Production task. *Frontiers in Human Neuroscience*. doi:10.3389/fnhum.2013.00711

Berk, L. E. (S. U. (2009). *Child Development* (8th editio). Pearson Education, Inc.

Cho, H. S. & Woo, T. H. (2016). Mechanical analysis of flying robot for nuclear safety and security control by radiological monitoring. *Annals of Nuclear Energy*, *94*, 138–143. doi:10.1016/j.anucene.2016.03.004

Chu, G., Hong, J., Jeong, D.-H., Kim, D., Kim, S., Jeong, S., & Choo, J. (2014). The experiments of wearable robot for carrying heavy-weight objects of shipbuilding works. *Automation Science and Engineering (CASE), 2014 IEEE International Conference*, 978–983. Retrieved from http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true%7B%5C&%7Dtp=%7B%5C&%7Darnumber=6899445%7B%5C&%7Durl=http://ieeexplore.ieee.org/iel7/6892922/6899294/06899445.pdf?arnumber=6899445

Clements, J. (2015). *Neural Networks Applied to the Discrete Sequence Production Task* (Doctoral dissertation, University of Strathclyde, University of Twente).

Gaudiello, I., Zibetti, E., Lefort, S., Chetouani, M., & Ivaldi, S. (2016). Trust as indicator of robot functional and social acceptance. An experimental study on user conformation to iCub answers. *Computers in Human Behavior*, *61*, 633–655. doi:http://dx.doi.org/10.1016/j.chb.2016.03.057

Gleitman, H., Gross, J., & Reisberg, D. (2011). *Psychology* (8th editio). New York, London: W.W. Norton & Company, Inc.

Grey, M. & Joo, S. (2014). Planning Heavy Lifts for Humanoid Robots *.

Gupta, A. & Noelle, D. (2007). A dual-pathway neural network model of control relinquishment in motor skill learning. *Proceedings of the International Joint Conference on ...* 405–410. Retrieved from http://www.aaai.org/Papers/IJCAI/2007/IJCAI07-063.pdf

Hebb, D. O. (1949). *The Organization of Behavior*. New York: NY: Wiley.

Hélie, S., Proulx, R., & Lefebvre, B. (2011). Bottom-up learning of explicit knowledge using a Bayesian algorithm and a new Hebbian learning rule. *Neural Networks*, *24*(3), 219–232.

Hélie, S., Roeder, J. L., Vucovich, L., Rünger, D., & Ashby, F. G. (2015). A neurocomputational model of automatic sequence production. *Journal of cognitive neuroscience*, *27*(7), 1456–1469.

Hikosaka, O., Sakai, K., Lu, X., Nakahara, H., Rand, M. K., Nakamura, K., ... Doya, K. (1999). Parallel neural networks for learning sequential procedures. *Trends in Neurosciences*, *22*(10), 464–471. doi:10.1016/S0166-2236(99)01439-3

Jackendoff, R. (2002). *Foundations of Language: Brain, Meaning, Grammar, Evolution*. New York: Oxford University Press Inc. doi:10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3

Jager, J. & Meijering, R. (2015). *iCub Robot*. University of Twente. Retrieved from https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1295

Kalat, J. W. (C. S. U. (2009). *Biological Psychology* (Tenth). Belmont: Nelson Education, Ltd.

Liu, M. & Padois, V. (2015). Reactive whole-body control for humanoid balancing on non-rigid unilateral contacts, 3981–3987. doi:10.1109/IROS.2015.7353938

Metta, G., Sandini, G., Vernon, D., Natale, L., & Nori, F. (2008). The iCub humanoid robot: an open platform for research in embodied cognition. *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, 50–56. doi:http://dx.doi.org/10.1145/1774674.1774683

Mohd, N., Hamizah, S., Fikry, A., Musa, R., Ahmad, S. S., Mara, U. T., ... Mara, U. T. (2014). Autism Children : Cost and Benefit Analysis of Using Humanoid in Malaysia, 185–187.

Mori, M., MacDorman, K. F., & Kageki, N. (2012). The uncanny valley. *IEEE Robotics and Automation Magazine*, *19*(2), 98–100. doi:10.1109/MRA.2012.2192811

Panzer, S., Muehlbauer, T., Krueger, M., Buesch, D., Naundorf, F., & Shea, C. H. (2009). Effects of interlimb practice on coding and learning of movement sequences. *Quarterly journal of experimental psychology (2006)*, *62*(7), 1265–1276. doi:10.1080/17470210802671370

Pennisi, P., Tonacci, A., Tartarisco, G., Billeci, L., Ruta, L., Gangemi, S., & Pioggia, G. (2015). Autism and social robotics: A systematic review. *Autism Research*, (October 2015), 165–183. doi:10.1002/aur.1527

Ramirez-Amaro, K., Beetz, M., & Cheng, G. (2015). Understanding the intention of human activities through semantic perception: observation, understanding and execution on a humanoid robot. *Advanced Robotics*, *29*(5), 345–362. doi:10.1080/01691864.2014.1003096

Sakai, K., Hikosaka, O., & Nakamura, K. (2004). Emergence of rhythm during motor learning. *Trends in Cognitive Sciences*, *8*(12), 547–553. doi:10.1016/j.tics.2004.10.005

Sciutti, A., Rea, F., & Sandini, G. (2014). When you are young, (robot's) looks matter. Developmental changes in the desired properties of a robot friend. *Proceedings - IEEE International Workshop on Robot and Human Interactive Communication*, *2014-Octob*(October), 567–573. doi:10.1109/ROMAN.2014.6926313

Tsagarakis, N. G., Metta, G., Sandini, G., Vernon, D., Beira, R., Becchi, F., ... Caldwell, D. G. (2007). iCub: the design and realization of an open humanoid platform for cognitive and neuroscience research. *Advanced Robotics*, *21*(10), 1151–1175. doi:10.1163/156855307781389419

van der Velde, F. (2015). *Outline of a motor neural blackboard*. University of Twente.

van der Velde, F. (2016). Concepts and Relations in Neurally Inspired In Situ Concept-Based Computing. *Frontiers in Neurorobotics*, *10*(May), 1–6. doi:10.3389/fnbot.2016.00004

van der Velde, F. & de Kamps, M. [M.]. (2015). Combinatorial structures and processing in neural blackboard architectures. In T. R. Besold, A. d'Avila Garcez, G. F. Marcus, & R. Miikkulainen (Eds.), *Proceedings of the workshop on cognitive computation: integrating neural and symbolic approaches (coco@nips 2015)* (pp. 1–9). Montreal: CEUR Workshop Proceedings.

van der Velde, F. & de Kamps, M. [Marc]. (2006). Neural blackboard architectures of combinatorial structures in cognition. *The Behavioral and brain sciences*, *29*(1), 37–70, discussion 70–108. doi:10.1017/S0140525X06009022

Verwey, W. B. (2003). Processing modes and parallel processors in producing familiar keying sequences. *Psychol. Res. 67*, 106–122.

Verwey, W. B. (2010). No TitleDiminished motor skill development in elderly: Indications for limited motor chunk use. *Acta Psychol. 134*, 206–214.

Wilson, H. R. & Cowan, J. D. (1972). Excitatory and inhibitory interactions in localized populations of model neurons. *Biophysical journal*, *12*(1), 1–24. doi:10.1016/S0006-3495(72)86068-5

Yun, S.-S., Kim, H., Choi, J., & Park, S.-K. (2015). A robot-assisted behavioral intervention system for children with autism spectrum disorders. *Robotics and Autonomous Systems*, *76*, 58–67. doi:10.1016/j.robot.2015.11.004

Appendices

A iCub installation tutorial

UNIVERSITEIT TWENTE

HUMAN FACTORS & ENGINEERING PSYCHOLOGY

MASTER'S THESIS

Tutorial iCub Software

Author:
Janina Roppelt
s1194526

First Supervisor:
Prof. Dr. Frank van der Velde
Second Supervisor:
Prof. Dr. Ing. Willem Verwey

June 29, 2016

Contents

1 Goal and prerequisites

This tutorial is for students that have basic knowledge of python and are interested in working with the iCub. Therefore it is expected that you know about the components used to work with the iCub.
Knowledge about how computer systems interact with each other can help to understand what we will be doing, but is not mandatory.
You should know what a command line interface is and how you can start it on your operating system. If you feel unsure about that visit `http://www.davidbaumgold.com/tutorials/command-line/` for more information.

This tutorial will help you to set up everything you need. Be sure to read it carefully, especially when you have to choose an option. As a rough guideline, choices are indicated by bullet points and actions are indicated by numbers.

Do you feel confident to begin? Then let's start!

2 Installing the iCub software

In this section you will learn how to install all necessary software to communicate with the iCub as well as the iCub simulator.

Two parts will be described:

- Installation of a virtual machine
- Installation on Linux (Ubuntu)

If you have another operating system than Linux, you first have to install the virtual machine, otherwise you can begin with section 2.2, the iCub software installation on Linux.
There are ways to run the software on Windows or Mac but they are more complex and very prone to errors. Another reason to choose Linux is that the iCub itself also works with an Ubuntu distribution. So when not using Linux already, you can become familiar with this operating system through the virtual machine. This is a safe way, as even in the unlikely event of you breaking something important, it will not affect your original operating system.
Should you have a really good reason to install the software on Windows or Mac anyway, please visit `http://wiki.icub.org/wiki/ICub_Software_Installation` for more information.

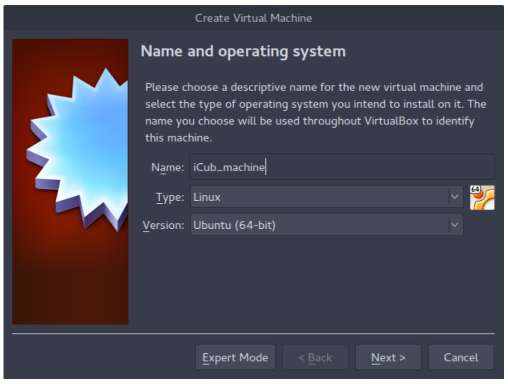
2.1 Installation of a virtual machine

In case you don't have Linux running on your device already, you should install a virtual machine. The first step you have to take is installing a program that makes and runs virtual machines. We chose VirtualBox as virtualizer.

1. Download VirtualBox from `https://www.virtualbox.org/wiki/Downloads`. Choose the version for your operating system.
2. Install VirtualBox. You can use the default options, you don't have to change anything.

Next you have to download the image of the Ubuntu operating system and integrate it into VirtualBox.

1. Download the Xenial 64bit version (VirtualBox (VDI) 64bit) from `http://www.osboxes.org/ubuntu-gnome/`.
2. If you don't have a 7z zipping program, download and install it from `http://www.7-zip.org/download.html`.
3. Unzip the image in a directory of your choice. It should be a directory you can find back, but not one that is used frequently and where you could delete the file by accident. An example is the VagrantBox directory.
4. Open VirtualBox.
5. Click the "New" button in the upper left corner.
6. In the resulting window:
   - Choose a name, for example "iCub_machine"
   - Choose Linux as type
   - Choose Ubuntu (64-bit) as version

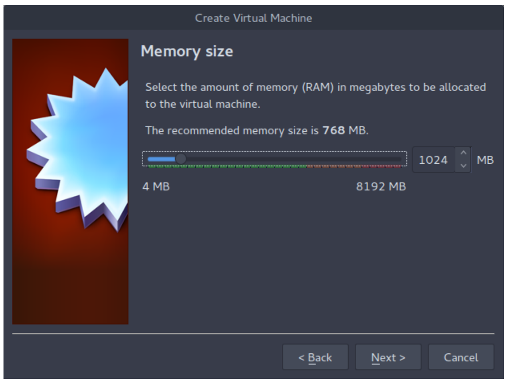

Create Virtual Machine

Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: iCub_machine
Type: Linux
Version: Ubuntu (64-bit)

7. Click on next.
8. Set the slider to 1024 MB and click on next.


Create Virtual Machine

Memory size

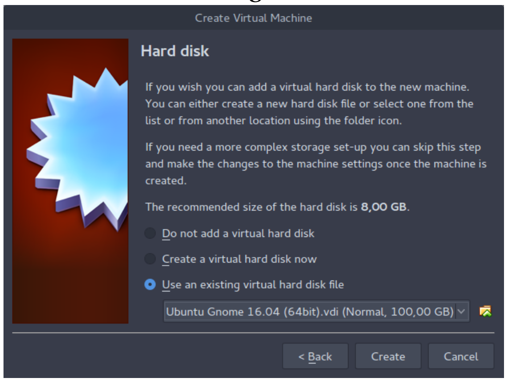
Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **768** MB.

1024 MB

4 MB 8192 MB

9. Choose the option “Use an existing virtual hard disk file” and navigate (click on the folder in the lower right corner) to the file you just extracted.


Create Virtual Machine

Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard disk is **8,00 GB**.

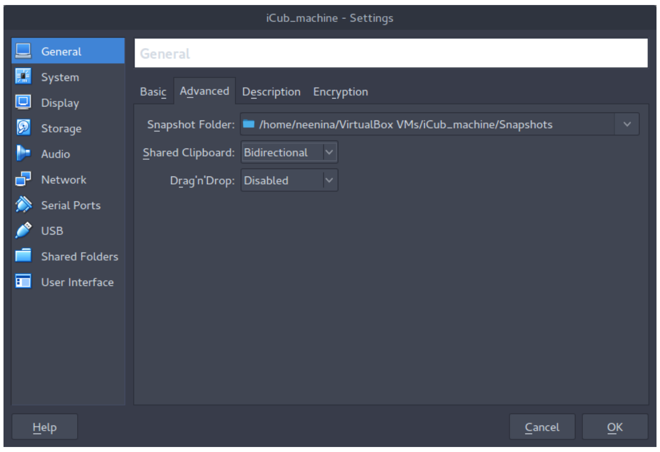
- Do not add a virtual hard disk
- Create a virtual hard disk now
- [✓ Use an existing virtual hard disk file]

Ubuntu Gnome 16.04 (64bit).vdi (Normal, 100,00 GB)

10. Click create.

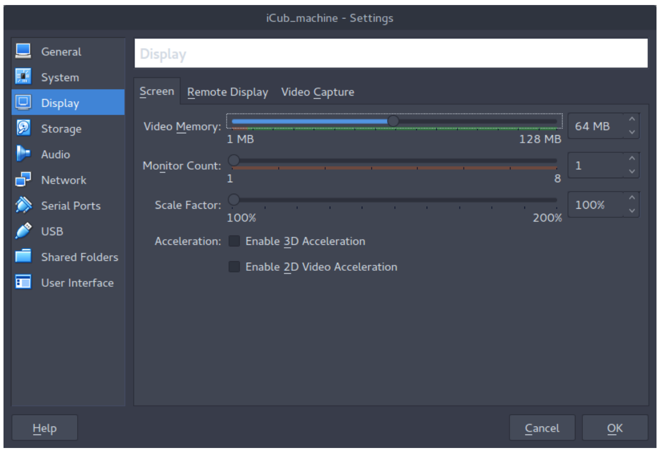
Now the virtual machine is installed, we will make some changes to increase the workability of the system.

1. Select the machine you just made and click on settings (can also be done via a right click and then settings).
2. In the “General” section, go to the tab “Advanced” and set Shared Clipboard to “Bidirectional”. That will enable you to copy-paste from your own operating system to the virtual machine (VM) and vice-versa.



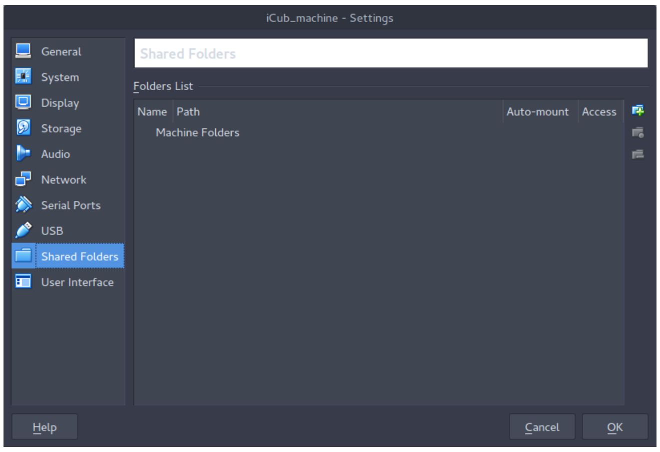
Snapshot Folder: /home/neenina/VirtualBox VMs/iCub_machine/Snapshots
Shared Clipboard: Bidirectional
Drag'n'Drop: Disabled

3. In the "Display" section (Screen tab) set the Video Memory to 64 MB. If you don't do that your virtual machine will be rather slow.



Video Memory: 64 MB
Monitor Count: 1
Scale Factor: 100%
Acceleration: Enable 3D Acceleration
Enable 2D Video Acceleration

4. OPTIONAL: In the "Shared Folders" section you can add a map that can be used by both, your original OS and the VM. This might be handy to transfer project files easily.



5. Click on OK.

6. Start the virtual machine with a double click or by pressing the green “start” arrow.

7. At the login, your password is “osboxes.org”.

8. At the upper left corner, in the “Devices” tab, click on “install guest settings” (last item) and confirm the installation.

9. After the installation is finished, restart the machine. Do so by clicking on the battery symbol in the upper right corner. Then, in the pop-up window go to the lower right corner and click on the power symbol and choose “Restart”.

10. Log in again. Now your resolution should be fitted to the resolution of your screen and your VM installation and configuration is done.

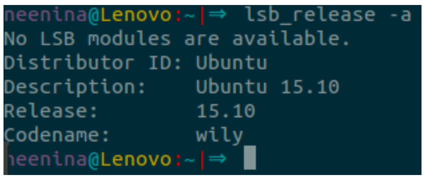
Now, you have a suitable operating system for the iCub software. To install this software, follow the following part of the tutorial.

2.2 Installation on Ubuntu

Now we will install the iCub software components. First, you have to check your Linux distribution. If you just installed a virtual machine, you can skip this part

1. Open your terminal. For example: press the Windows-key, type terminal and hit enter.

2. Check which version of Ubuntu you’re running by typing “lsb_ release -a” You will get a list like this:



```
neenina@Lenovo:~|⇒  lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 15.10
Release:        15.10
Codename:       wily
neenina@Lenovo:~|⇒
```

The part we are interested in is the codename. Remember it or write it down.

3. Check if you can install the software from pre-compiled binaries through visiting `http://www.icub.org/ubuntu/dists/`. If your codename is in the list you are ready to go.

If your version is not supported yet, use the virtual machine option. You can check from while to while if they finished the new compilation. To give you an idea of the time dimensions: for Xenial - the most recent version at this moment - it took approximately one month. For wily - the previous one - it took around five months.

If your version is supported go back to you terminal.

NOTE: If you don't use Xenial, you will have to substitute "Xenial" with the code name of your own distribution

1. Execute

```
sudo sh -c 'echo "deb http://www.icub.org/ubuntu xenial contrib/science" > /etc/
    apt/sources.list.d/icub.list'
```

   This will add the iCub repository to your source list.

2. To update your list of packages execute

```
sudo apt-get update
```

3. Finally, to install the software execute

```
sudo apt-get install icub
```

   NOTE: the packages are not signed, so you can't know if they have been tampered with. I personally had no problems but you should decide for yourself if you trust the site as well as the connection enough to install the packages.

Congratulations! You know have all the tools you need to facilitate communication with the iCub.

3 First encounter with the iCub software

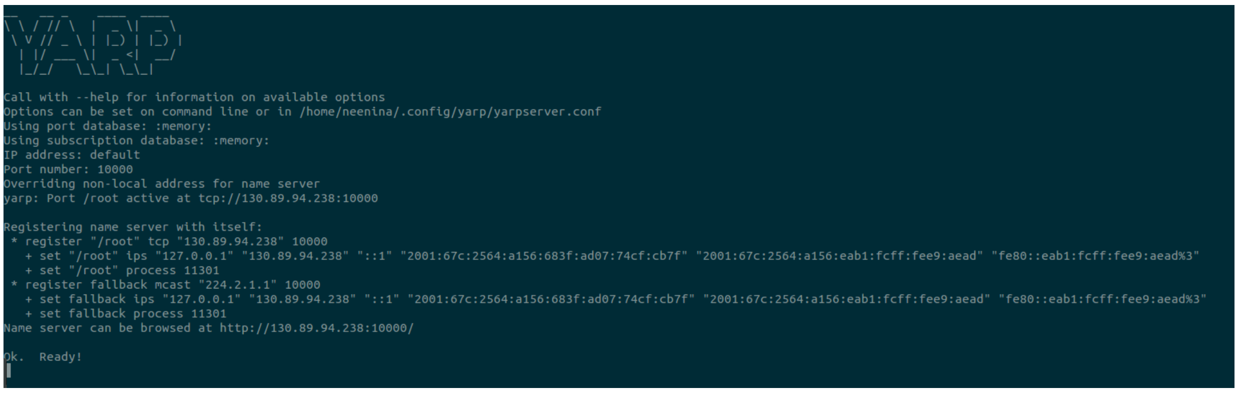
Now all your programs are working, let's see what we can do with them.

First of all let's start and test the YARP server.

1. Open a terminal
2. In that terminal type

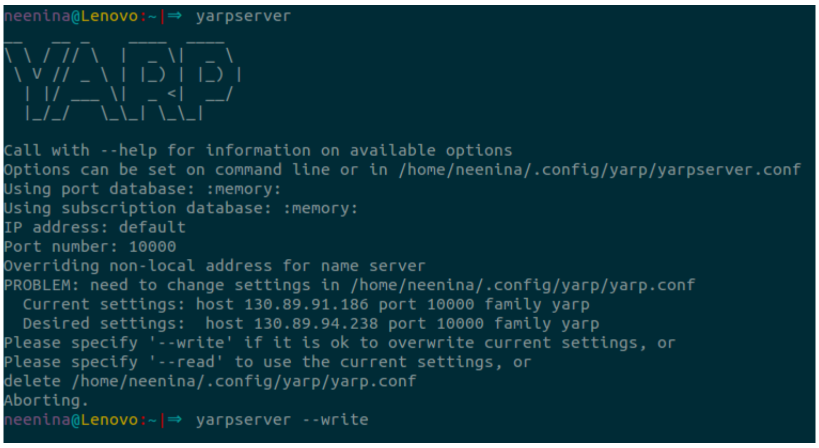
```
yarpserver
```

This will start the YARP server and you will get an output like this:



```
__   __ _    ____  ____
\ \ / // \  |  _ \|  _ \
 \ V // _ \ | |_) | |_) |
  | |/ ___ \|  _ <|  __/
  |_/_/   \_\_| \_\_|

Call with --help for information on available options
Options can be set on command line or in /home/neenina/.config/yarp/yarpserver.conf
Using port database: :memory:
Using subscription database: :memory:
IP address: default
Port number: 10000
Overriding non-local address for name server
yarp: Port /root active at tcp://130.89.94.238:10000

Registering name server with itself:
 * register "/root" tcp "130.89.94.238" 10000
   + set "/root" ips "127.0.0.1" "130.89.94.238" "::1" "2001:67c:2564:a156:683f:ad07:74cf:cb7f" "2001:67c:2564:a156:eab1:fcff:fee9:aead" "fe80::eab1:fcff:fee9:aead%3"
   + set "/root" process 11301
 * register fallback mcast "224.2.1.1" 10000
   + set fallback ips "127.0.0.1" "130.89.94.238" "::1" "2001:67c:2564:a156:683f:ad07:74cf:cb7f" "2001:67c:2564:a156:eab1:fcff:fee9:aead" "fe80::eab1:fcff:fee9:aead%3"
   + set fallback process 11301
Name server can be browsed at http://130.89.94.238:10000/

Ok.  Ready!
```

NOTE: If you get an error looking like this:

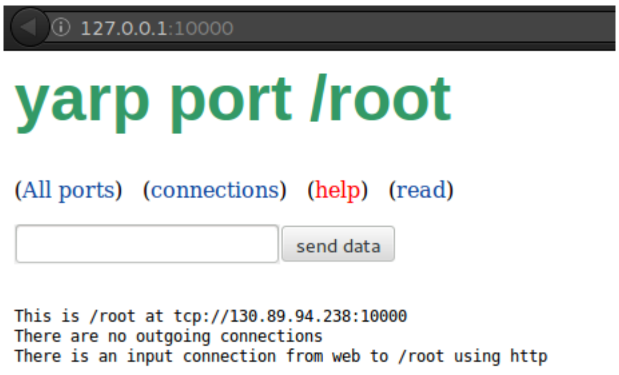


```
neenina@Lenovo:~|⇒  yarpserver
__   __ _    ____  ____
\ \ / // \  |  _ \|  _ \
 \ V // _ \ | |_) | |_) |
  | |/ ___ \|  _ <|  __/
  |_/_/   \_\_| \_\_|

Call with --help for information on available options
Options can be set on command line or in /home/neenina/.config/yarp/yarpserver.conf
Using port database: :memory:
Using subscription database: :memory:
IP address: default
Port number: 10000
Overriding non-local address for name server
PROBLEM: need to change settings in /home/neenina/.config/yarp/yarp.conf
  Current settings: host 130.89.91.186 port 10000 family yarp
  Desired settings:  host 130.89.94.238 port 10000 family yarp
Please specify '--write' if it is ok to overwrite current settings, or
Please specify '--read' to use the current settings, or
delete /home/neenina/.config/yarp/yarp.conf
Aborting.
neenina@Lenovo:~|⇒  yarpserver --write
```

Don't panic. You might use wifi on the campus which results in your IP address being changed between sessions. Execute the yarpserver command again, but now with the write option

```
yarpserver --write
```

3. To see information about the name server in your browser, got to `http://127.0.0.1:10000`. It should look like this:



```
This is /root at tcp://130.89.94.238:10000
There are no outgoing connections
There is an input connection from web to /root using http
```

Now we know for sure that our server is up and running! So let us start to communicate. In the following, don't touch the terminal we just use to start the server. It is now running a process and trying to work with that terminal might stop the process or will give you unexpected behaviour. This is also true for the other programs we will start in this section. Therefore:

1. Open three other terminals.
2. In the first terminal execute

```
yarp read /portread
```

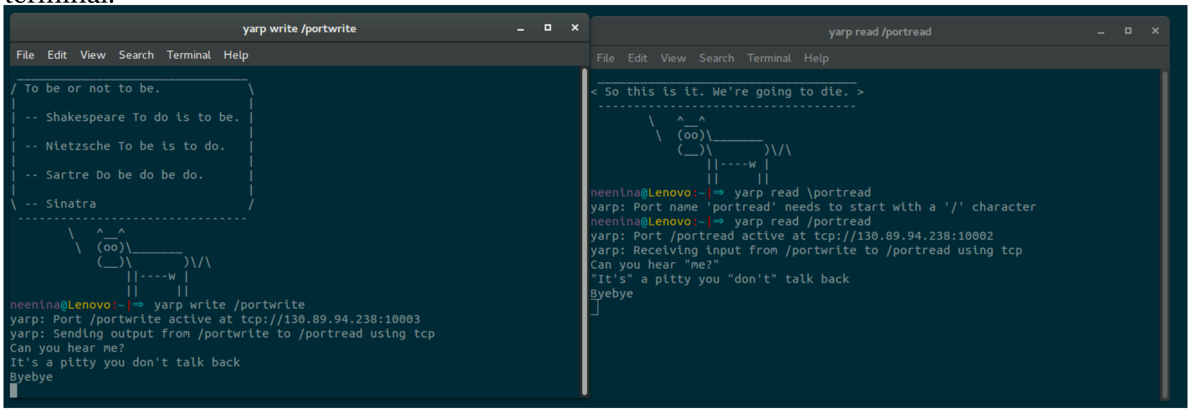
3. In the second terminal execute

```
yarp write /portwrite
```

4. In the third terminal execute

```
yarp connect /portwrite /portread
```

5. You have now started two clients that can communicate with each other. To test your set-up just type anything you want into the write terminal and see it appear in the read terminal.


yarp write /portwrite
File Edit View Search Terminal Help
yarp read /portread
File Edit View Search Terminal Help

```
 _________________________________
/ To be or not to be.             \
|                                 |
| -- Shakespeare To do is to be.  |
|                                 |
| -- Nietzsche To be is to do.    |
|                                 |
| -- Sartre Do be do be do.       |
|                                 |
\ -- Sinatra                      /
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
neenina@Lenovo:~|⇒ yarp write /portwrite
yarp: Port /portwrite active at tcp://130.89.94.238:10003
yarp: Sending output from /portwrite to /portread using tcp
Can you hear me?
It's a pitty you don't talk back
Byebye
```

```
 ____________________________________
< So this is it. We're going to die. >
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
neenina@Lenovo:~|⇒ yarp read \portread
yarp: Port name 'portread' needs to start with a '/' character
neenina@Lenovo:~|⇒ yarp read /portread
yarp: Port /portread active at tcp://130.89.94.238:10002
yarp: Receiving input from /portwrite to /portread using tcp
Can you hear "me?"
"It's" a pitty you "don't" talk back
Byebye
```

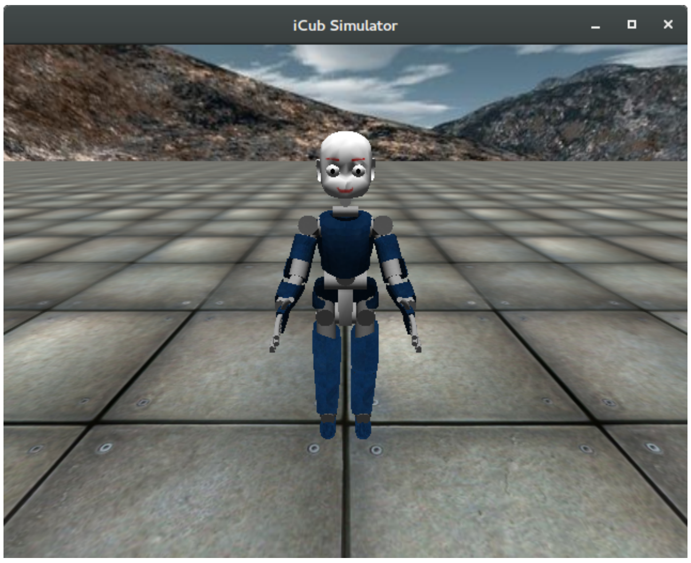
That wasn't to hard, was it?
You can close all terminals except the yarpserver itself (the first terminal we used to run the server by typing yarpserver). The three extra terminals were only for testing the server and therefore can be closed, now that know that it works.
We go on with starting the iCub simulator. To do so

1. Open a new terminal

2. Start the simulator by executing

```
iCub_SIM
```

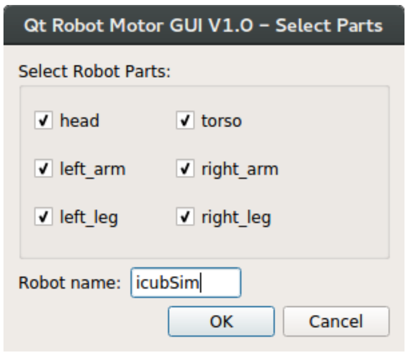
Now we have a little model of our robot standing in a generated environment:



3. One of the possibilities to control the simulator is the YARP-motor-GUI. In a new terminal, start the GUI by executing

```
yarpmotorgui
```

4. In the resulting window:



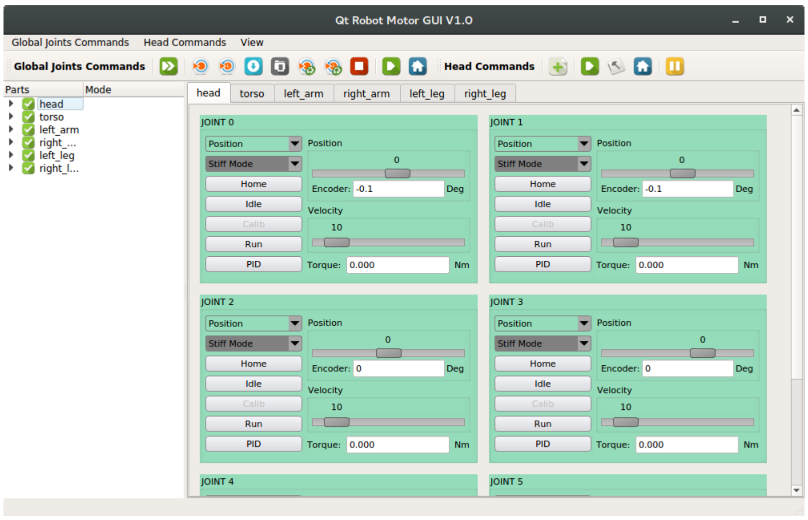
Qt Robot Motor GUI V1.0 – Select Parts

Select Robot Parts:

- [✓] head
- [✓] torso
- [✓] left_arm
- [✓] right_arm
- [✓] left_leg
- [✓] right_leg

Robot name: icubSim

Use “icubSim” as name. It is important to do that because the program won’t find the simulator otherwise. Now you see the GUI which can be used to move the iCub:



5. Now just try what happens if you change the levers. Also pay attention to the tabs at the top. You cannot only control the head but also chose an arm for example. Can you make the iCub wave?

“But what about python?” you might ask. We will start on that part in the following section.

4 Integrate python with YARP

In this section you will learn how to establish a communication line between Python and the YARP server that controls all communication to and from the iCub (-Simulator).

To follow this part of the tutorial you should have installed python (python2.7) and the python developer options (python2.7-dev). If you haven't done that yet you can do it now for Windows via `https://www.python.org/downloads/` or for Ubuntu via your command line by executing

```
sudo apt-get install python2.7 python2.7-dev
```

Because this will make changes to your system you are asked to give your password. If you have both python and the iCub software you can start with the bindings.

This part of the tutorial involves a lot of command line operations. Generally it is good if you either don't get output, or if you get back a lot of information. In case that a step failed, check if you wrote everything correctly.

To install the bindings we need two programs, SWIG and Cmake. SWIG is a software development tool that connects C and C++ code with high-level programs like Python. We need that because nearly all software for the iCub is written in a C language. Cmake is a compiling tool that is needed to execute the code which will make the bindings. For more information on both programs visit `swig.org` and `cmake.org` respectively.

1. Begin by installing SWIG and Cmake through executing

   ```
   sudo apt-get install swig cmake
   ```

   You might get the feedback that you already have these programs. In that case just continue.

2. Go to `https://github.com/robotology/yarp/releases` and download the "Source code (tar.gz) " by clicking on it.
   We use the source code of YARP here, because the version we already have is optimized for binaries (the software was compiled for us) and can therefore not be used to compile the bindings. If you would compile the software yourself, you could use the source code you already have. In this tutorial the bindings are used because they are straight forward to install and less prone to errors.

3. Unpack the downloaded tar.gz file in a directory of your choice.

4. Open that directory in your terminal

5. Change to the subdirectory "bindings" by executing

```
cd bindings
```

6. Make a new directory “build” by running

```
mkdir build
```

7. Change to the new directory through

```
cd build
```

8. Execute

```
cmake ../ -DCREATE_PYTHON=ON
```

   This step will fit the installation to your system. It automatically uses your build tool, python version and the place for the installation

9. Transform the source code to python modules by running

```
make
```

10. At last execute

```
export PYTHONPATH=$PYTHONPATH:`pwd`
```

   This tells your system where you installed the bindings so that python can use them. ATTENTION: ‘pwd‘ is your current location, so this command will only work if you haven’t changed your directory in the process. If you followed the instructions closely that should not be a problem.
   The path you need is: “directory_where_you_unpacked/bindings/build”

Now let’s test if the installation worked. To do that

1. Start python by typing

```
python
```

2. Test the bindings through executing

```
import yarp
```

If there is no error you are done with this step and can go on to the next part of this tutorial.

If you get an error message, delete the unpacked files, go back to step three of this part and redo all the following steps carefully.

Do you still experience problems? It is not very likely, but it can happen that there is a problem in the source code you downloaded. Go back to step 2 but choose an older version this time.

5 Write your first program to control the iCub

Now that you have everything in place to make the iCub move with your own python programs. In this part of the tutorial I will explain the general idea of how this is done. What we will do is simple, we want a sequence of finger movements. The finger movements should be in such a form that they can be extended into movements one would expect while playing piano.
So let's start:

1. Start the yarpserver, the iCub simulator, and the YARP-motor-GUI. If you forgot how to do that look back at the "First encounter with the iCub software" part.
   NOTE: If some later steps don't work, make sure the yarpserver and the iCub simulator are running. The yarpserver is running if there is output in the terminal in which you started it. If not used for some time it will state: "Name server running happily" The iCub simulator is running if you can see it. In case that problems continue while you can't find any mistakes, it can also help to restart both, the server and the simulator.

2. Open a python editor *NOTE:* Depending on your editor, you may also have to add your pythonpath to your environment settings for YARP to work. The path is the location of the build folder you made in the bindings part. As this is a really specific step I can't explain it in depth. It might help to google the name of your editor plus import YARP.

3. In a new file, set up the communication with the iCub like this:

```
import time      # import the time module, will be needed later
import yarp      # import the YARP bindings. Depending on your editor, you may also
     have to add your pythonpath to your environment settings for YARP to work.
    The path is the location of the build folder you made in the bindings part

yarp.Network.init()      # initialise YARP

# prepare a property object
props = yarp.Property()
props.put("device", "remote_controlboard")
props.put("local", "/client/left_arm")       # if we would like to control another
    body part left_arm should be the name of that body part
props.put("remote", "/icubSim/left_arm")      # icubSim is the name of the simulator
    . When communicating with the real robot, another name should apply

# create remote driver
armDriver = yarp.PolyDriver(props)

# query motor control interfaces
iPos = armDriver.viewIPositionControl()
iEnc = armDriver.viewIEncoders()

# retrieve number of joints. These are identical to the joints in the yarpmotorgui
jnts=iPos.getAxes()

# print the number of joints to control if everything is working
```

```
print 'Controlling', jnts, 'joints'

# read encoders. This will give you the current position of all parts of the arm
encs = yarp.Vector(jnts)
while not(iEnc.getEncoders(encs.data())):
    time.sleep(0.1) # This while-loop is important because the program will
    otherwise move on too quickly and not all positions will be read correctly. If
     you forgot to do this, you may encounter strange behaviour like a sudden
    movement at the beginning of your program an no real return to the home
    position

# store the current position as home position
home = yarp.Vector(jnts, encs.data())
# If you want to control your home position, run the following loop, to print all
    encoders. Then check if they are the same as they were in the motorgui
for i in xrange(16):
    print home[i], i

# initialize a new tmp vector identical to encs. This vector will be used to move
    the fingers, while not loosing the home position
tmp = yarp.Vector(jnts, encs.data())
```

Listing 1: Set everything up

4. Now we have everything we need to begin with the actual movement:

```
def movefinger(finger):     # a function to move a finger to a specific location
    and back
    if finger == 0:     # finger 0 is the index finger
        tmp.set(11, 30) # this function moves joint 11 by 30 units. I tried which
    joints I have to use and how much to move it by playing with the motorgui
    if finger == 1:     # finger 1 is the middle finger
        tmp.set(13, 30) # same as above but now with the middle finger joint
    if finger == 2:     # finger 2 are both the ring and little finger, they seem
    to have only one motor
        tmp.set(15, 30)
    iPos.positionMove(tmp.data())    # the above can be seen as settings, this is
    the actual movement


def returnfinger(finger):    # a function to return a finger to the home position
    iPos.positionMove(home.data())  # this moves the fingers back
    if finger == 0:
        tmp.set(11, 0)  # here we set each finger back to 0, to be able to use tmp
     again later
    if finger == 1:
        tmp.set(13, 0)
    if finger == 2:
        tmp.set(15, 0)

# now we have everything we need to let the iCub move! Just call the functions
movefinger(0)
time.sleep(1)   # wait one second. This is done to give the simulator time to
    actually move
```

```
returnfinger(0)
time.sleep(1)
movefinger(1)
time.sleep(1)
returnfinger(1)
time.sleep(1)
movefinger(2)
time.sleep(1)
returnfinger(2)
```

Listing 2: Let the fingers move!

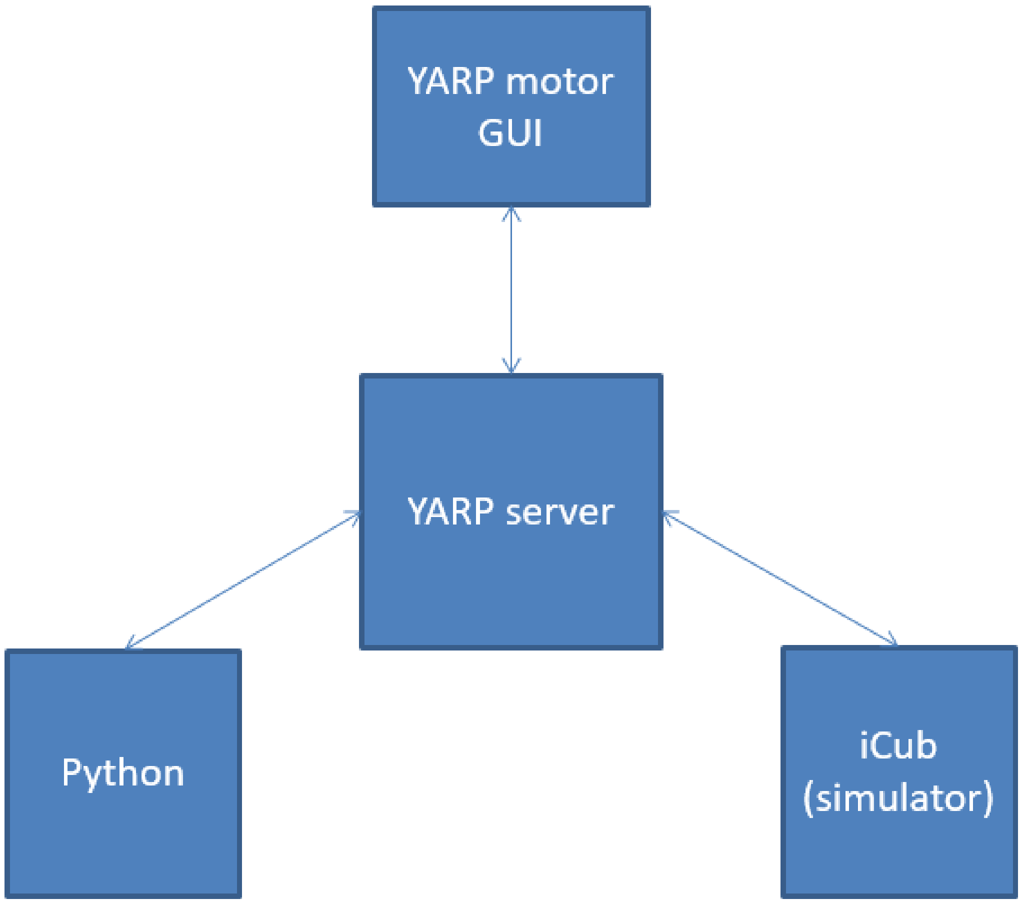
5. Try to let the fingers move in another order. Also see what happens if you only call one of the functions, if you call the same one two times after each other, or if you leave out the sleep function.

Now you know how to move fingers, can you get the iCub to wave again? This time with the right hand and through a short program?

You also should feel familiar enough with how to move the iCub around using python. You have successfully completed this tutorial!

6 Extra information about the the name of the iCub

This part of the tutorial will help you understand how to handle the “name” of the iCub as well as the simulator. First let me tell you in short what the iCub network looks like.



In the picture you can see the three components. Your Python program, the iCub (simulator) and the motor-GUI, all talking to the yarp server. The server can be seen as the man in the middle, distributing information the components. Think about calling someone with your phone. You will need a number to do that. And so does the iCub. The “number” of the iCub is its name and therefore ensures that everybody knows who he’s talking to. Of course the other components also have some kind of identifier, but as there are few cases where it would be needed to do that we will not consider them.

First of all, let’s learn how to give a name to our simulator.

1. Open a terminal
2. Start the YARP - server by executing

```
yarpserver
```

3. Open another terminal

4. In that terminal, execute the command

```
iCub_SIM --name icub
```

   The first part of the command starts the iCub simulator, while the second is telling it to start with a name, *icub* in this case. The default name for the simulator, when not using the –name option is “iCubSim”. The name *icub* can be exchanged with any other name, as long as it is one word.

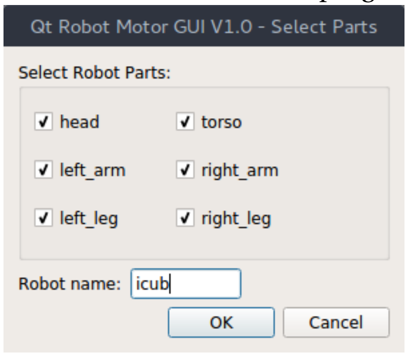
Now we have given the iCub simulator a name, let’s look at how we can tell the motor GUI this name.

1. Open a third terminal

2. Execute

```
yarpmotorgui
```

3. In the start window of the program, use “icub” as name



Qt Robot Motor GUI V1.0 - Select Parts

Select Robot Parts:

[✓] head [✓] torso
[✓] left_arm [✓] right_arm
[✓] left_leg [✓] right_leg

Robot name: icub

   Do you see why I chose the name *icub* when you start the motor-GUI yourself?

Finally, the following steps will teach you how to communicate with the iCub from your python program.

1. Consider the program we used above to move the fingers

2. The important line can be found at the setup of the connection:

```
props.put("remote", "/icubSim/left_arm")
```

"icubSim" is the default name of the simulator, that is why we used it here.

3. Now that out robot is called "icub" we have to change this line, so the program knows where to "reach" the iCub.

```
props.put("remote", "/icub/left_arm")
```

So when thinking about the real iCub, let's call him Philip, it should be no problem for you to know how you would connect him to Python and the YARP-motor-GUI now, right?

B Code

B.1 Rhythm.py

```
"""
Simulation to learn rhythms to be used in the motor−blackboard simulation
"""
import time
import json

# Input device
# 0 = keyboard
# 1 = iCub
input_device = 0

# Ask for name of the song to learn
song = raw_input('What song should I learn? (name of the song) −−>')

# Define how to save the rhythm (song.txt in the S−node directory)
rhy_in_dir = 'Snodes/' + song + '.txt'

# Ask for number of positions
sn = int(raw_input('How many positions does the rhythm have? −−>'))

# Ask for number of repetitions
train = int(raw_input('How often should I play? −−>'))

'''
Import necessary modules based on input device
'''
if input_device == 0:
    import pygame
elif input_device == 1:
    import icub
'''
Calculate amount of chunks the rhythm should have, based on the amount of positions
'''
amount = (sn − (sn % 4)) / 4 + 1
if sn % 4 == 0:
    amount −= 1

'''
Make a new rhythm array filled with zeros
Use eight positions per chunk −> two values per S−node: the first is the begin time of a
    S−node, the second the end time
Use as many chunks as calculated based on all positions in the rhythm
Append a zero to save how often the rhythm was learned
'''
rhythm = [[0] ∗ 8 for i in xrange(amount)]
rhythm.append([0])

'''
Set variables to manage loops
'''
# number of values that have to be saved (begin and end time times positions)
s = sn ∗ 2

t = 0
snode = 0
learn = 0
```

```
'''
Initialize pygame if pygame has to be used
'''
if input_device == 0:
    pygame.init()
    screen = pygame.display.set_mode((640, 480))

    # Outer while loop: redo learning loop as long as there are runs left
    # First inner while loop: wait until enter is pressed
    # Second inner while loop: one iteration of learning a rhythm
    while train > 0:
        print "Press enter to start"
        t = 0
        snode = 0
        wait = True
        while wait:
            pygame.event.pump()
            pressed = pygame.key.get_pressed()
            if pressed[pygame.K_RETURN] == 1:
                wait = False

        # Set start time of learning loop as reference
        tim = time.time()*1000

        print "Start rhythm learning"

        # Go on while there are positions to fill
        # Fill rhythm from left to right
        # rhythm[snode] is the chunk of the rhythm that is active
        # rhythm[snode][t % 8] is the position of a chunk the loop is in
        # snode is increased after 8 steps (begin and end time for 4 positions)
        while t < s:
            pygame.event.pump()
            pressed = pygame.key.get_pressed()

            # If space is pressed an t is even, save begin time of S-node
            if pressed[pygame.K_SPACE] == 1 and t % 2 == 0:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time()*1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1

            # If space is released an t is uneven, save end time of S-node
            if pressed[pygame.K_SPACE] == 0 and t % 2 == 1:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time()*1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1

        # increase number of times rhythm was learned
        rhythm[-1][0] += 1
        train -= 1

# If input device is the iCub, do same as above with the difference that not key presses
    are monitored, but the angles of one of the iCubs finger
# icub.get_Pos is called to receive the angles of the joints of the robot
elif input_device == 1:
    while train > -1:
        print "Move the index finger to start"
        t = 0
        snode = 0
        wait = True
        while wait:
            if icub.get_Pos(0) > 15:
                wait = False
        tim = time.time() * 1000
        print "Start rhythm learning"
        while t < s:
            if icub.get_Pos(0) > 30 and t % 2 == 0:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time() * 1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1
            if icub.get_Pos(0) < 10 and t % 2 == 1:
                if rhythm[-1] == [0]:
                    rhythm[snode][t % 8] = (time.time() * 1000 - tim)
                else:
                    rhythm[snode][t % 8] = (rhythm[snode][t % 8] +
                                            (time.time() * 1000 - tim)) / 2
                t += 1
                if t % 8 == 0 and t > 0:
                    snode += 1
        rhythm[-1][0] += 1
        train -= 1

'''
Calculate delta of the first position and normalize array so that the rhythm begins at
    100.
'''
delta = rhythm[0][0] - 100
for x in xrange(amount):
    for y in xrange(8):
        rhythm[x][y] -= delta

'''
Calculate theoretical and actual length of the array in order to calculate where the
    learned rhythm ends
'''
theo = amount * 8
real = s
numb = theo - real
end = rhythm[amount - 1][7-numb]

'''
Fill up the rest of the array with dummy values, 50 higher than the endpoint.
As they are all the same, no S-nodes will be activated by these dummies (no time in
    between begin and end time)
'''
```

```
for k in xrange(8):
    if -delta - 1 < rhythm[amount - 1][k] < -delta + 1:
        rhythm[amount-1][k] = end + 50

'''
Save file to predefined location
'''
with open(rhy_in_dir, 'w') as outfile:
    json.dump(rhythm, outfile)

print 'I\'m done, the rhythm looks like this\n', rhythm, '\nLet\'s play!'
```

Listing 21: rhythm.py

B.2 Motorblackboard.py

```python
"""
motorBlackboard_2016.py
Version 23−6−2016.
Based on motor BB program 6−5−2015.
Here: simulation in numpy matrix form.
Population dynamics: Wilson Cowan.
Wilson Cowan: two combined populations: E (excitatory) and I (inhibitory).
Dynamics in "motorblackboard_dynamics.py"
Population dynamics calculated with 4th order Runge Kutta numerical integration.
Populations can receive external input.
Input assumed to be constant during each step in Runge Kutta.
"""
import json
import pygame
from random import random

import numpy as np
import pygame.event

import motorblackboard_dynamicsc as mb
import wav

import matplotlib.pyplot as plt
import csv

'''
Configurations
'''
# Devices to be used:
# 0 = Automatic
# 1 = Keyboard
# 2 = iCub as input
# 3 = iCub as output of automatic simulation
use = 2

# The factor the rhythm is multiplied with
# Simulation is "faster" with a low number
speed = 2

'''
Configurations for cognition and speed af learning
'''
# Begin weight long term potential
bw = 0.05

# Begin chance to learn right
bcg = 0.7

# How quick the program learns
learning_factor = 1.01

# Activation needed to learn
learning_threshold = 30

'''
Do not change
'''
ri = 0
startc = 0
```

```
LTPU = 0
song = 'rhtyhm'
songc = 'rhythmc'
f = 0
fnew = 0
cn = 0
LTPL = []

'''
Make run specifications with user input
'''
new = raw_input('Would you like me to learn a new song or play a learned one? (new,
    learned) -->')
if new == 'new':
    LTPU = 1
    songs = raw_input('What song should I learn? (name of the song if a rhythm exists or
     train for training rhythm) -->')
    song = 'Snodes/' + songs + '.txt'
    songc = 'Cnodes/' + songs + 'c.txt'
    fnew = int(raw_input('How many different notes does the song have? -->'))

if LTPU == 0:
    songs = raw_input('Which song do you want to play? -->')
    song = 'Snodes/' + songs + '.txt'
    songc = 'Cnodes/' + songs + 'c.txt'

runs = int(raw_input('How often should I play? -->'))

# Import iCub module when necessary
if use == 2 or use == 3:
    import icub

# import rhythm from file
with open(song, 'r') as outfile:
    r = json.load(outfile)

# number of snodes in rhythm
sn = len(r[ri]) / 2

for a in xrange(len(r) - 1):
    for b in xrange(len(r[a])):
        r[a][b] *= speed

# calculate simulation duration
n = r[-2][- 1] + 500  # number of steps in an iteration, determined through rhythm *last
    rhythm value + 100
n = int(n)

# import LTP from file
if LTPU == 0:
    with open(songc, 'r') as outfile:
        LTPL = json.load(outfile)
    cn = len(LTPL)
    f = len(LTPL[0][0])
elif LTPU == 1:
    cn = len(r) - 1  # number of cnodes
    f = fnew

# Calculate number of gates
gate = sn * f
```

```
# Print relevant data
print "Rhythm:", songs
print "Number of fingers:", f
print 'Number of gates:', gate
print "Number of C-nodes:", cn
print "Numbers of S-nodes per chunk:", sn

"""
Chunk nodes.
Cnode_in = Input from Cnode_e into gate column (to pop3)
Cnode_e (Cnode_i) is matrix: first index is n (time),
second index is number of Cnode.
"""
x1 = np.zeros(n * cn)
y1 = x1.reshape(n, cn)
Cnode_e = y1.copy()
Cnode_i = y1.copy()

Cnode_in = 0.0

# List of cnodes to check wheater the right finger was played
# Has to be filled in by hand
# This list is for playing smoke on the water
list_cnodes = [[0] * 4 for i in xrange(cn)]
list_cnodes[0][0] = 0
list_cnodes[0][1] = 1
list_cnodes[0][2] = 2
list_cnodes[0][3] = 0

list_cnodes[1][0] = 1
list_cnodes[1][1] = 3
list_cnodes[1][2] = 2
list_cnodes[1][3] = 0

list_cnodes[2][0] = 1
list_cnodes[2][1] = 2
list_cnodes[2][2] = 1
list_cnodes[2][3] = 0

# print list_cnodes
globc = startc

# LTP
LTP = [0]

Cnode_l = np.zeros((sn, f))

# LTP_flags = Cnode_l.copy() * cn # used to limit learning to one time per trial

Cnode_l = [[0] * f for i in xrange(sn)]

LTP_flags = [[[0] * f for xy in xrange(sn)] for gf in xrange(cn)]

LTPN = [[[bw] * f for xx in xrange(sn)] for gg in xrange(cn)]

if LTPU == 0:
    LTP = LTPL
elif LTPU == 1:
    LTP = LTPN

if len(LTP) > len(r):
    print "That won't work, your rhythm is too short for your sequence"
    exit()

# Chance
chance_l = np.zeros((sn, f))
chance = chance_l.copy()
bcf = (1.0 - bcg) / f - 1
cf = [0] * cn

if cn == 0 or f == 0:
    print "you did something wrong"
    exit()
"""
Sequence nodes.
Snode_e (Snode_i) is matrix: first index is n (time),
second index is number of Snode.
"""
x = np.zeros(n * sn)
y = x.reshape(n, sn)

Snode_e = y.copy()
Snode_i = y.copy()

"""
Feedback nodes.
Feedback from, e.g., fingers (actuators)
To be used later
Fback is matrix: first index is n (time),
second index is number of Fback.
"""
x = np.zeros(n * f)
y = x.reshape(n, f)

Fback = y.copy()

"""
Populations in gate column
See motor sequence model.pptx or "Outline motor model.pdf" Fig 3
p1 = population 1 in gate column, etc
Wilson Cowan: each population consist of an excitatory (e.g., p1_e)
and inhibitory sub population (p1_i).
p1_e is matrix: first index is n (time),
second index is number of gate.
Same for other populations
"""
x = np.zeros(n * gate)
y = x.reshape(n, gate)

p1_e = y.copy()
p1_i = y.copy()
p2_e = y.copy()
p2_i = y.copy()
p3_e = y.copy()
p3_i = y.copy()
p4_e = y.copy()
p4_i = y.copy()
p5_e = y.copy()
p5_i = y.copy()
p6_e = y.copy()
p6_i = y.copy()
p7_e = y.copy()
```

```
p7_i = y.copy()
p8_e = y.copy()
p8_i = y.copy()
p9_e = y.copy()
p9_i = y.copy()

"""
External (global)input on population 7 (gate_i) of all gating columns:
"""
Inh_chunk = 20.0

"""
Define activation function for main simulation run:

Activate C-node
Deactivate C-node

Activate S-node with C-node
Activate S-node
Deactivate S-node

Activate Feedback
Deactivate Feedback

How activation works:
mb.pop_step_wc_m(i, 0, Cnode_e, Cnode_i, 20.0, 20.0):
first input (i) is time
Second input (0) is order (1e, 2e, etc) of Cnode (or Snode)
Third and fourth input (Cnode_e, Cnode_i,): the Wilson Cowan populations
Fifth and sixth input (20, 20): input for the Wilson Cowan populations

"""



# Activate C-node
# Arguments: C-node to activate, rhythm to use
# Call calculate_chance if automatic input is used and chance is not calculated yet
# cf[0] is a flag to prevent repeated calculation of the chance. Set in calculate_chance
    and deactivate_cnode
# Call rhythm function on the given arguments
# Call mb.pop_step_wc_m to activate given C-node if the simulation is still running
# r[-2][-1] is the valid value in the rhythm array. The if is added because the
    activation wouldn't stop otherwise
def activate_cnode(c, rit):
    if cf[c] == 0:
        calculate_chance(c)
    rhythm(rit, c)
    if i < r[-2][- 1] + 150:
        return mb.pop_step_wc_m(i, c, Cnode_e, Cnode_i, 20.0, 20.0)


# Deactivate C-node
# Argument: C-node to deactivate
# Use global variables globc, ri, and cf
# Global variables have to be defined because they are changed within the function but
    have to change outside, too
# Sets flag cf[globc] to 0 so chance can be calculated again in the next run
# If cf[c] == 1, the function is called for the first time
# In that case, if there are still C-nodes left in the song, change global used C-node
    and rhythm to the next one
# Call amb.pop_step_wc_m with value 0 to deactivate C-node
```

```
def deactivate_cnode(c):
    global globc
    global ri
    global cf
    if cf[c] == 1:
        cf[c] = 0
        if globc < len(LTP) - 1:
            globc += 1
            ri += 1
    return mb.pop_step_wc_m(i, c, Cnode_e, Cnode_i, 0.0,
                            0.0)


# Make S-node flag
snodeflag = [0] * sn


# Activate S-node with C-node
# Arguments: S-node to activate, C-node to activate it with
# Uncomment if loop to print when a S-node is active exactly once (visual feedback)
# Call mb.pop_step_wc_m to activate given S-node with the present activation of the given
     C-node
def activate_snode_with_cnode(snode, cnode):
    #if snodeflag[snode] == 0:
        #print 'S-node', snode + 1, 'active'
        #snodeflag[snode] = 1
    return mb.pop_step_wc_m(i, snode, Snode_e, Snode_i, 0.2 * Cnode_e[i, cnode], 0.2 *
    Cnode_e[
        i, cnode])


# Activate S-node
# Argument: S-node to activate
# Uncomment if loop to print when a S-node is active exactly once (visual feedback)
# Call mb.pop_step_wc_m to activate given S-node with standard activation
def activate_snode(snode):
    #if snodeflag[snode] == 0:
        #print 'S-node', snode + 1, 'active'
        #snodeflag[snode] = 1
    return mb.pop_step_wc_m(i, snode, Snode_e, Snode_i, 20.0, 20.0)


# Deactivate S-node
# Set flag back to 0
# Call amb.pop_step_wc_m with value 0 to deactivate S-node
def deactivate_snode(snode):
    if snodeflag[snode] == 1:
        snodeflag[snode] = 0
    return mb.pop_step_wc_m(i, snode, Snode_e, Snode_i, 0.0, 0.0)


# Make feedback flag
fbackflag = [0] * f


# Activate Feedback
# Argument: Finger that gives feedback
# Set feedback of the given finger (i = present step of the simulation) to 100 (max)
# Uncomment if loop to print when feedback is active exactly once (visual feedback)
# Return feedback
def feedback_on(finger):
    Fback[i][finger] = 100
    #if fbackflag[finger] == 0:
        #print 'Feedback from finger:', finger + 1
        #fbackflag[finger] = 1
    return Fback[i][finger]


# Deactivate Feedback
# Argument: Finger of which feedback has to be turned off
# Set feedback of the given finger (i = present step of the simulation) to 0
# Set flag back to 0
# Return feedback
def feedback_off(finger):
    Fback[i][finger] = 0
    if fbackflag[finger] == 1:
        fbackflag[finger] = 0
    return Fback[i][finger]


"""
Define logic functions for the main simulation:

Run rhythm (def rhythm)

Constantly check if learning situation is given (def active_learning)

Learn
"""



# Run rhythm
# Arguments: rhythm to be used, active C-node
# r is the rhythm array
# r[rhyt] is the present active chunk of the rhythm
# r[rhyt][0] is the start time of the first S-node r[rhyt][1] its end time. resp. for
    other S-nodes
# Call activate_snode_with_cnode on the first S-node between the first two positions of
    the rhythm
# Call deactivate_snode in the break between first and second S-node
# Call activate_snode for all following S-nodes if time is a begin time (even index)
# Call deactivate_snode for all following S-nodes if time is an end time (uneven index)
# If the last step of the rhythm is reached, call deactivate_snode on last S-node,
    # and if there is a C-node left call activate_cnode on this C-node
# If last S-node time is exceeded by 150, deactivate used C-node (done later to leave
    last S-node time to deactivate)
def rhythm(rhyt, cnode):
    if r[rhyt][0] < i < r[rhyt][1]:
        activate_snode_with_cnode(0, cnode)
    if r[rhyt][1] < i < r[rhyt][2]:
        deactivate_snode(0)
    for w in xrange(len(r[rhyt]) - 1):
        if r[rhyt][w] < i < r[rhyt][w + 1] and w % 2 == 1 and w != 1:
            deactivate_snode(w / 2)
        if r[rhyt][w] < i < r[rhyt][w + 1] and w % 2 == 0 and w != 0:
            activate_snode(w / 2)
    if i > r[rhyt][- 1]:
        deactivate_snode(sn - 1)
        if cnode < len(r) - 2:
            activate_cnode(globc + 1, ri + 1)
    if i > r[rhyt][- 1] + 150:
        deactivate_cnode(globc)
```

```
# Check if learning condition is given
# Search for active C-node
# If C-node is active, search for active S-node
# If S-node is active, search for gates in which:
    # feedback is given (pop6 active),
    # pop3 activation is about the learning threshold,
    # and no flag has been set yet
# If such a gate exists, call learn function with found C-node, S-node and gate as
    arguments
def activate_learning():
    for cnode in xrange(cn):
        if Cnode_e[i, cnode] > 99:
            for snode in xrange(sn):
                if Snode_e[i, snode] > 95:
                    for finger in xrange(gate):
                        if p6_e[i, finger] > 95 and p3_e[i, finger] > learning_threshold
    and LTP_flags[cnode][snode][finger % f] == 0:
                                    learn(cnode, snode, finger % f)


# Update LTP (learn)
# Arguments: Active C-node, active S-node, active finger
# Set LTP to present value times the learning_factor
# LTP[cnode] is a matrix with as much positions as gates
# LTP[cnode][pos][finger] is the weight between the C-node and the gate (S-node and
    finger define the gate)
# list_cnodes is a list of which finger has to be played in which position based on the C
    -node
# Print correctness of learning
# Set flag of gate to 1 so learning is only done once per run
def learn(cnode, pos, finger):
    LTP[cnode][pos][finger] *= learning_factor
    if finger == list_cnodes[cnode][pos]:
        print "I learned right!"
    else:
        print "I learned wrong :("
    LTP_flags[cnode][pos][finger] = 1


# Calculate chance for automatic learning
# Argument: C-node to calculate the chance for
# Use global variable cf
# Set flag on cf
# bcg is the begin chance of the good finger to be used, set in the beginning of the
    program
# bcf is the begin chance of a wrong finger to be learned, calculated from bcg and the
    amount of fingers
# list_cnodes is a list of which finger has to be played in which position based on the C
    -node
# LTP[cnode][pos] are all weights a C-node has at one position of the sequence
# LTP[cnode][pi][xf] is the weight between the C-node and the gate (S-node and finger
    define the gate)
# chance[pi][xf] chance of a finger (xf) to be activated in a position (pi)
# For every position in the C-node:
    # look up the good finger,
    # calculate the sum of all weights
    # make temporary array for chance per finger
    # for every finger:
        # calculate the weighted weight of that finger
        # calculate difference between weighted weight and expected weight (can also be
    negative)
        # add/substract difference from begin chances to correct for learned associations
     and store it in the chance array
def calculate_chance(cnode):
    global cf
    cf[cnode] = 1
    for pi in xrange(sn):
        good = list_cnodes[cnode][pi]
        sumw = sum(LTP[cnode][pi])
        c = [0]*f
        for xf in xrange(f):
            c[xf] = (LTP[cnode][pi][xf])/sumw
            c[xf] = c[xf] - 1.0/f
            chance[pi][xf] = (bcg if xf == good else bcf)+c[xf]


"""
Set rules of activation based on inout:
Automatic

keyboard

iCub
"""


# Manage automatic feedback, does at the moment only work with three fingers
# Arguments: random numbers for every finger
# Feedback is activated 50 steps after the S-node and deactivated 50 steps before the
    deactivation of the S-node
# r[ri][0] is the first position of the active rhythm
# chance[0][1] is the chance that finger two is activated at the first position. In the
    beginning the chance is 15%
# If none of the "wrong" choices is valid, the "right" feedback is activated
def feedback_automatic(ran1, ran2, ran3):
    if i > r[ri][0] + 50:
        if ran1 < chance[0][1]:
            feedback_on(1)
        elif ran1 > 1.0 - chance[0][2]:
            feedback_on(2)
        else:
            feedback_on(0)

    if i > r[ri][1] - 50:
        if ran1 < chance[0][1]:
            feedback_off(1)
        elif ran1 > 1.0 - chance[0][2]:
            feedback_off(2)
        else:
            feedback_off(0)

    if i > r[ri][2] + 50:
        if ran2 < chance[1][0]:
            feedback_on(0)
        elif ran2 > 1.0 - chance[1][2]:
            feedback_on(2)
        else:
            feedback_on(1)

    if i > r[ri][3] - 50:
        if ran2 < chance[1][0]:
            feedback_off(0)
        elif ran2 > 1.0 - chance[1][2]:
            feedback_off(2)
        else:
            feedback_off(1)

    if i > r[ri][4] + 50:
        if ran3 < chance[2][1]:
            feedback_on(1)
        elif ran3 > 1.0 - chance[2][0]:
            feedback_on(0)
        else:
            feedback_on(2)

    if i > r[ri][5] - 50:
        if ran3 < chance[2][1]:
            feedback_off(1)
        elif ran3 > 1.0 - chance[2][0]:
            feedback_off(0)
        else:
            feedback_off(2)


# Manage feedback from keyboard
# Assign keys to fingers and call feedback_on if the key is pressed
# Get all pressed keys through pygame event
# If a key is pressed, call feedback_on on respective finger
# If a key is released, call feedback_off on respective finger
def feedback_key():
    pygame.event.pump()
    pressed = pygame.key.get_pressed()

    if pressed[pygame.K_1] == 1:
        feedback_on(0)
    else:
        feedback_off(0)

    if pressed[pygame.K_2] == 1:
        feedback_on(1)
    else:
        feedback_off(1)

    if pressed[pygame.K_3] == 1:
        feedback_on(2)
    else:
        feedback_off(2)

    if pressed[pygame.K_4] == 1:
        feedback_on(3)
    else:
        feedback_off(3)

    # uncomment when using more than four fingers
    #if pressed[pygame.K_5] == 1:
        #feedback_on(4)
    #else:
        #feedback_off(4)


# Define how iCub data should be interpreted
# Arguments: minimun and maximum angle
# call function icub.get_Pos to receive angle parameter
# If angle is bigger than the max value, turn feedback on that finger on
# if finger is returned under the minimal value, turn feedback off
def feedback_iCub(maxi, mini):
    if icub.get_Pos(0) > maxi:
        feedback_on(0)

    if icub.get_Pos(0) < mini:
        feedback_off(0)

    if icub.get_Pos(1) > maxi:
        feedback_on(1)

    if icub.get_Pos(1) < mini:
        feedback_off(1)

    if icub.get_Pos(2) > maxi:
        feedback_on(2)

    if icub.get_Pos(2) < mini:
        feedback_off(2)

    if icub.get_Pos(3) > maxi:
        feedback_on(3)

    if icub.get_Pos(3) < mini:
        feedback_off(3)



"""
Define how output of the simulation should be displayed
"""
# Make gate activation flag
finger_flag = [0] * gate


# Manage the output of the simulation
# If there is a gate with a pop2 activation above 90, calculate the corresponding finger
    # Call feedback_on on that finger
    # If there was no output action on this finger yet (fingerflag[fingers] == 0):
        # if the iCub is used, call icub_movefinger
        # call wav.play_note with the finger and the song as argument
        # Set flag to 1
# If a finger has less than an activation of 10
    # turn feedback off
    # if the iCub is used, call icub_returnfinger
    # set flag back to 0
def manage_output():
    for fingers in xrange(gate):
        finger = fingers % f
        if p2_e[i, fingers] > 90.00:
            feedback_on(finger)
            if finger_flag[fingers] == 0:
                if use == 2 or use == 3:
                    icub.movefinger(finger)
                print "Finger activated: ", finger + 1
                wav.play_note(finger, songs)
                finger_flag[fingers] = 1
        if 90 > p2_e[i, fingers] > 0.1:
            feedback_off(finger)
            if use == 2 or use == 3:
                if finger_flag[fingers] == 1:
                    icub.returnfinger(finger)
            finger_flag[fingers] = 0


"""
Main simulation run

Based on the input, the simulation is either started automatic or with an action

For each run the feedback method is defined and learning and output is activated

Activation simulation starts at time = 100 (t100) with the activation of the first C-node
"""
if __name__ == '__main__':
    for bla in xrange(runs):
        Fback = y.copy()

        # random determines a number which will determine if learning will be right.
        ran1 = random()
        ran2 = random()
        ran3 = random()

        globc = startc
        ri = startc

        # Let every not automatic simulation run begin with an action
        if use == 1:
            pygame.init()
            screen = pygame.display.set_mode((640, 480))
            wait = True
            print "Press enter to start"
            while wait:
                pygame.event.pump()
                pressed = pygame.key.get_pressed()
                if pressed[pygame.K_RETURN] == 1:
                    wait = False
        if use == 2:
            wait = True
            print "Move a finger to start"
            while wait:
                if icub.get_Pos(0) > 11:
                    wait = False

        for x in cf:
            cf[x] = 0
        print "Start of the simulation"
        for i in xrange(n - 1):

            # choose which input version is used:
            if use == 0 or use == 3:
                feedback_automatic(ran1, ran2, ran3)
            if use == 1:
                feedback_key()
            if use == 2:
                feedback_iCub(30, 10)

            # activate learning and output
            activate_learning()
            manage_output()

            # if i > 100 and simulation is not at the end, activate appropriate C-node
            if i > 100:
                if globc < len(LTP):  # if there still is a C-node left
                    activate_cnode(globc, ri)  # also activates rhythm

            """
            Calculation of the populations in the gating columns
            See "Outline motor model.pdf" Fig 3
            Input for a gate column:
            Snode_e: input from the excitatory population of the sequence node connected
    to the gate
            Cnode_in: input from the Chunk nodes, determined in the for loop
            Fback: feedback from the fingers (actuators in general)
            Inh_chunk: global input to give constant activation for pop7
            """
            for k in xrange(gate):
                if k < f:
                    position = 0
                elif k < 2 * f:
                    position = 1
                elif k < 3 * f:
                    position = 2
                else:
                    position = 3
                Cnode_in = 0.0
                finger = k % f
                Cnode_in += LTP[globc][position][finger] * Cnode_e[i, globc]
                mb.gate_column(i, k, finger, position, Snode_e, Cnode_in, Fback,
                               Inh_chunk, p1_e, p1_i, p2_e, p2_i, p3_e, p3_i, p4_e, p4_i,
                               p5_e, p5_i, p6_e, p6_i, p7_e, p7_i, p8_e, p8_i, p9_e, p9_i
    )

        print "End of the simulation\nUpdated LTP:", LTP

    with open(songc, 'w') as outfile:
        json.dump(LTP, outfile)

    """
    Calculation of reaction times (RT)
    Here: calculate the first time that activation of output population
    p2_e exceeds 90.0
    See Fig3 in document "Outline motor model.pdf"
    """
    RT = np.zeros(gate)

    for j in xrange(n):
        for k in xrange(gate):
            if RT[k] == 0.0:
                if p2_e[j, k] > 90.0:
                    RT[k] = j

    """
    Print results

    Make selections by commenting
    dif = difference between plotcurves with same values
    vary dif at will for clarity
    """
```

```
print "First reaction per gate: ", RT

x_axis = np.linspace(0, n, n)  # for the activation plots

dif = 5.0  #

"""
Printing populations in all gates
Print C-nodes and  S-nodes
Print feedback
col is number of instance (variable)
Differ curves by adding factor + x*dif + y*col

for col in xrange(gate):
    plt.plot(x_axis, p1_e[:, col] + 1 * dif + 2 * col)
    plt.plot(x_axis, p2_e[:, col] + 3*dif + 2*col)
    plt.plot(x_axis, p3_e[:, col] + 5*dif + 2*col)

    plt.plot(x_axis, p4_i[:, col] + 4*dif + 2*col, 'c--')
    plt.plot(x_axis, p5_i[:, col] + 5*dif + 2*col, 'b--')
    plt.plot(x_axis, p6_i[:, col] + 6 * dif + 2 * col, 'm--')
    plt.plot(x_axis, p7_i[:, col] + 7*dif + 2*col, color='green')
    plt.plot(x_axis, p8_i[:, col] + 7*dif + 2*col)
    plt.plot(x_axis, p9_i[:, col] + 6*dif + 2*col)


for col in xrange(cn):
    plt.plot(x_axis, Cnode_e[:, col] + 12 * dif, 'blue')

for col in xrange(sn):
    plt.plot(x_axis, Snode_e[:, col], color='magenta')

for col in xrange(f):
    plt.plot(x_axis, Fback[:, col])

# print specific activations with set colors and line style, add legend
S1, = plt.plot(x_axis, Snode_e[:, 0], color='magenta')
S2, = plt.plot(x_axis, Snode_e[:, 1], color='#00FF00')
S3, = plt.plot(x_axis, Snode_e[:, 2], color='red')
S4, = plt.plot(x_axis, Snode_e[:, 3], color='cyan')
C1, = plt.plot(x_axis, Cnode_e[:, 0], 'b--')
C2, = plt.plot(x_axis, Cnode_e[:, 1], color='#0080FF', linestyle='--')
C3, = plt.plot(x_axis, Cnode_e[:, 2], color='black', linestyle='--')

plt.legend([S1, S2, S3, S4, C1, C2, C3], ['S-node 1', 'S-node 2', 'S-node 3', 'S-node
 4', 'C-node 1', 'C-node 2', 'C-node 3'], prop=fontP)
plt.ylim(0, 110)

plt.xlabel('Time (ms)')
plt.ylabel('Activity')
plt.title('Population activity in gating circuit motor BB')

plt.show()
plt.savefig('temp1.png')   #Print plot to file

"""

"""
Save data to .csv files for plotting and analysis

with open("csv/Snode.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(Snode_e)

with open("csv/Cnode.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(Cnode_e)

with open("csv/Fback.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(Fback)

with open(csv/"p1_e.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p1_e)

with open("csv/p2_e.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p2_e)

with open("csv/p3_e.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p3_e)

with open("csv/p4_i.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p4_i)

with open("csv/p5_i.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p5_i)

with open("csv/p6_i.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p6_i)

with open("csv/p6_i.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p7_i)

with open("csv/p8_i.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p8_i)

with open("csv/p9_i.csv", "wb") as f:
    writer = csv.writer(f)
    writer.writerows(p9_i)
"""
```

Listing 22: motorblackboard_2016.py

B.3 Motorblackboard_dynamicsc.py

```
"""
Version: 11-1-2016
Import file of functions needed to simulate dynamics in Motor Blackboard (MB)
See: Outline motor model.pdf", Fig 4.
Wilson Cowan: two combined populations: E (excitatory) and I (inhibitory).
Population dynamics calculated with 4th order Runge Kutta numerical integration.
Populations can receive external input.
Input assumed to be constant during each step in Runge Kutta.
"""

from ctypes import *

# Load library
lib = cdll.LoadLibrary("./libmotorblackboard.so")
lib.deriv_ex.restype = c_double
lib.deriv_in.restype = c_double
lib.pop_step_wc_m.restype = py_object

import numpy as np


"""
time constants tau_ex, tau_in for E, I. Already in the form of 1/tau.
constants for derivatives of E, I, and logistic function:
alfa, epsilonfor E
gamma, delta for I
beta, teta for logistic function
f_max = maximum activation population
n = Number of steps in for-loop of simulation run.
See Wilson Cowan dynamics, Fig 4 in "Outline motor model.pdf"
"""
tau_ex = 0.4
tau_in = 0.8
alfa = 0.2
epsilon = 0.20
gamma = 0.20
delta = 0.20
beta = 2
teta = 5
f_max = 100
w_all = 0.20

"""External variable for step size in 4th order Runge Kutta."""
h = 0.1

"""
Functions for calculating the derivatives.
E, I = excitatory, inbibitory populations. E,I are arrays.
Input = external input to E, I populations.
alfa, epsilon = constants for time derivative of E.
gamma, delta = constants for time derivative of I.
f_max = maximum activation for logistic function.
tau_ex, tau_in = time constants for E, I.
beta, teta = constants in logistic function (teta = threshold).
"""


def deriv_ex(E, I, Input):
    return lib.deriv_ex(c_double(E),c_double(I),c_double(Input))
```

```python
def deriv_in(E, I, Input):
    return lib.deriv_in(c_double(E),c_double(I),c_double(Input))


"""
Gradual decline of input.
value = input.
start = loop index of start decline
rate = rate of decline (float)
"""


def decline(i, start, value, rate):
    r = value * (1 - np.exp(-(rate * (i - start))))
    return r


"""
Function for calculating one step in 4th order Runge Kutta.
ips = index step in for loop of simulation run.
excit, inhib = array exc (inh) values of E, I.
exc_input, inh_input = array external input (assumed constant in each step).
h = step size, h is external variable.
Here: input not with array
pop_step_wc_m: WC calculation in matrix form
"""


def pop_step_wc_m(ips, k, excit, inhib, exc_input, inh_input):
    new_excit, newinhib = lib.pop_step_wc_m(c_double(excit[ips,k]), c_double(inhib[ips,k
    ]), c_double(exc_input), c_double(inh_input))
    excit[ips + 1, k] = new_excit
    inhib[ips + 1, k] = newinhib


"""
Function for simulating gating module.
See "motor sequence model.pptx", or "Outline motor model.pdf" (Fig 3)
Pop1 = population 1 etc.
"""


def gate_column(j, k, f, sn, Snode, Cnode, Fback, Inh_chunk_e, pop1_e, pop1_i,
                pop2_e, pop2_i, pop3_e, pop3_i, pop4_e, pop4_i, pop5_e, pop5_i,
                pop6_e, pop6_i, pop7_e, pop7_i, pop8_e, pop8_i, pop9_e, pop9_i):

    input1 = w_all * Snode[j, sn]
    pop_step_wc_m(j, k, pop1_e, pop1_i, input1, input1)

    input2 = w_all * pop1_e[j, k] - w_all * pop8_i[j, k]
    pop_step_wc_m(j, k, pop2_e, pop2_i, input2, input2)

    input3 = Cnode - w_all * pop7_i[j, k]
    pop_step_wc_m(j, k, pop3_e, pop3_i, input3, input3)

    input4 = w_all * pop1_e[j, k] - w_all * pop6_i[j, k]
    pop_step_wc_m(j, k, pop4_e, pop4_i, input4, input4)

    input5 = w_all * pop1_e[j, k] - w_all * pop4_i[j, k]
    pop_step_wc_m(j, k, pop5_e, pop5_i, input5, input5)

    input6 = w_all * Fback[j, f]
    pop_step_wc_m(j, k, pop6_e, pop6_i, input6, input6)

    input7 = Inh_chunk_e - w_all * pop5_i[j, k]
    pop_step_wc_m(j, k, pop7_e, pop7_i, input7, input7)

    input8 = w_all * pop1_e[j, k] - w_all * pop9_i[j, k]
    pop_step_wc_m(j, k, pop8_e, pop8_i, input8, input8)

    input9 = w_all * pop3_e[j, k]
    pop_step_wc_m(j, k, pop9_e, pop9_i, input9, input9)

    return pop1_e, pop1_i, pop2_e, pop2_i, pop3_e, pop3_i, pop4_e, pop4_i, pop5_e, pop5_i
    , pop6_e, pop6_i, pop7_e, pop7_i, pop8_e, pop8_i, pop9_e, pop9_i
```

Listing 23: motorblackboard_dynamicsc.py

B.4 Motorblackboard_dynamics.c

```c
#include <Python.h>
#include <stdio.h>
#include <math.h>

// time constants tau_ex, tau_in for E, I. Already in the form of 1/tau.
// constants for derivatives of E, I, and logistic function:
// alfa, epsilonfor E
// gamma, delta for I
// beta, teta for logistic function
// f_max = maximum activation population
// n = Number of steps in for-loop of simulation run.
// See Wilson Cowan dynamics, Fig 4 in "Outline motor model.pdf"

#define tau_ex 0.4
#define tau_in 0.8
#define alfa 0.2
#define epsilon 0.20
#define gamma 0.20
#define delta 0.20
#define beta 2.
#define teta 5.
#define f_max 100.
#define w_all 0.20

// External variable for step size in 4th order Runge Kutta.
#define h 0.1


// Functions for calculating the derivatives.
// E, I = excitatory, inhibitory populations. E,I are arrays.
// Input = external input to E, I populations.
// alfa, epsilon = constants for time derivative of E.
// gamma, delta = constants for time derivative of I.
// f_max = maximum activation for logistic function.
// tau_ex, tau_in = time constants for E, I.
// beta, teta = constants in logistic function (teta = threshold).


double deriv_ex(double E, double I, double Input){
    double r1 = alfa * E - epsilon * I + Input;
    double r2 = exp(-beta * (r1 - teta));
    double r3 = f_max / (1 + r2);
    return tau_ex * (-E + r3);
}


double deriv_in(double E, double I, double Input){
    double r1 = gamma * E - delta * I + Input;
    double r2 = exp(-beta * (r1 - teta));
    double r3 = f_max / (1 + r2);
    return tau_in * (-I + r3);
}

// Function for calculating one step in 4th order Runge Kutta.
// ips = index step in for loop of simulation run.
// excit, inhib = array exc (inh) values of E, I.
// exc_input, inh_input = array external input (assumed constant in each step).
```

```
PyObject* pop_step_wc_m(double excit, double inhib, double exc_input, double inh_input){
    double K1 = h * deriv_ex(excit, inhib, exc_input);
    double L1 = h * deriv_in(excit, inhib, inh_input);
    double K2 = h * deriv_ex(excit + 0.5 * K1, inhib + 0.5 * L1, exc_input);
    double L2 = h * deriv_in(excit + 0.5 * K1, inhib + 0.5 * L1, inh_input);
    double K3 = h * deriv_ex(excit + 0.5 * K2, inhib + 0.5 * L2, exc_input);
    double L3 = h * deriv_in(excit + 0.5 * K2, inhib + 0.5 * L2, inh_input);
    double K4 = h * deriv_ex(excit + K3, inhib + L3, exc_input);
    double L4 = h * deriv_in(excit + K3, inhib + L3, inh_input);
    double nexcit = excit + (1 / 6.0) * (K1 + 2 * K2 + 2 * K3 + K4);
    double ninhib = inhib + (1 / 6.0) * (L1 + 2 * L2 + 2 * L3 + L4);

  PyObject* tupleOne = Py_BuildValue("(dd)", nexcit, ninhib);

    return tupleOne;
}
```

Listing 24: motorblackboard_dynamics.c

B.5 iCub.py

```
import time as t
import yarp

yarp.Network.init()     # Initialise YARP

#joint 11, 13 en 15
# prepare a property object
props = yarp.Property()
props.put("device", "remote_controlboard")
props.put("local", "/client/right_arm")
props.put("remote", "/icub/right_arm")

# create remote driver
armDriver = yarp.PolyDriver(props)

#query motor control interfaces
iPos = armDriver.viewIPositionControl()
#iVel = armDriver.viewIVelocityControl()
iEnc = armDriver.viewIEncoders()


# retrieve number of joints
jnts=iPos.getAxes()

print 'Controlling', jnts, 'joints'

# read encoders
encs = yarp.Vector(jnts)
while not(iEnc.getEncoders(encs.data())):
    t.sleep(0.1)

# store as home position
home = yarp.Vector(jnts, encs.data())
for i in xrange(16):
    print home[i], i

# initialize a new tmp vector identical to encs
tmp = yarp.Vector(jnts, encs.data())


# Get positions of joints
# Arguments: finger to get the position of
# Translate finger to joint
# Get all joints, wait until every joint has a value
# Return requested value
def get_Pos(finger):
    joint = finger * 2 + 9
    if joint == 9:
        joint = 10
    pos = yarp.Vector(jnts)
    while not(iEnc.getEncoders(pos.data())):
        t.sleep(0.1)
    return pos[joint]


# used to only move finger once.
finger_activated = [0]*4


```

```
# Move a finger of the iCub
# Arguments: number of a finger
# If a finger has to be moved, set temporary array to new value and move finger to that
    position
def movefinger(finger):
    global finger_activated
    if finger == 0 and finger_activated[0] == 0:
        tmp.set(10, 80)
        finger_activated[0] = 1
        iPos.positionMove(tmp.data())
    if finger == 1 and finger_activated[1] == 0:
        tmp.set(11, 20)
        tmp.set(12, 50)
        finger_activated[1] = 1
        iPos.positionMove(tmp.data())
    if finger == 2 and finger_activated[2] == 0:
        tmp.set(13, 20)
        tmp.set(14, 40)
        finger_activated[2] = 1
        iPos.positionMove(tmp.data())
    if finger == 3 and finger_activated[3] == 0:
        tmp.set(15, 80)
        finger_activated[3] = 1
        iPos.positionMove(tmp.data())


# Return a finger of the iCub
# Arguments: number of a finger
# If a finger has to be returned, set temporary array to standard value and move finger
    to home position
def returnfinger(finger):
    global finger_activated
    if finger == 0 and finger_activated[0] == 1:
        tmp.set(10, 10)
        finger_activated[0] = 0
        iPos.positionMove(home.data())
    if finger == 1 and finger_activated[1] == 1:
        tmp.set(11, 10)
        tmp.set(12, 18)
        finger_activated[1] = 0
        iPos.positionMove(home.data())
    if finger == 2 and finger_activated[2] == 1:
        tmp.set(13, 10)
        tmp.set(14, 18)
        finger_activated[2] = 0
        iPos.positionMove(home.data())
    if finger == 3 and finger_activated[3] == 1:
        tmp.set(15, 10)
        finger_activated[3] = 0
        iPos.positionMove(home.data())

'''
movefinger(1)
t.sleep(20)
returnfinger(1)
t.sleep(2)


movefinger(0)
t.sleep(30)
returnfinger(0)
```

```
t.sleep(2)

movefinger(1)
t.sleep(2)
returnfinger(1)
t.sleep(2)
movefinger(2)
t.sleep(2)
returnfinger(2)


tmp=yarp.Vector(jnts)


#tmp.set(11, tmp.get(11)+30)
#tmp.set(1, tmp.get(1)+10)
#tmp.set(2, tmp.get(2)+10)
#tmp.set(3, tmp.get(3)+10)



# move to new position
#iPos.positionMove(tmp.data())

#time.sleep(5)

iPos.positionMove(home.data())
'''
```

Listing 25: icub.py

B.6 wav.py

```python
import pygame
import time

pygame.init()


# Map fingers to notes
# First if statements define the song
# Second if statements set sound per finger
def play_note(finger, song):
    if song == 'smoke':
        if finger == 0:
            sound = pygame.mixer.Sound("g3.wav")
            sound.play()

        if finger == 1:
            sound = pygame.mixer.Sound("a4B.wav")
            sound.play()

        if finger == 2:
            sound = pygame.mixer.Sound("c4.wav")
            sound.play()

        if finger == 3:
            sound = pygame.mixer.Sound("c4B.wav")
            sound.play()
    if song == '7nation':
        if finger == 0:
            sound = pygame.mixer.Sound("B2.wav")
            sound.play()

        if finger == 1:
            sound = pygame.mixer.Sound("C3B.wav")
            sound.play()

        if finger == 2:
            sound = pygame.mixer.Sound("D3.wav")
            sound.play()

        if finger == 3:
            sound = pygame.mixer.Sound("E3.wav")
            sound.play()

        if finger == 4:
            sound = pygame.mixer.Sound("G3.wav")
            sound.play()
    if song == 'ente':
        if finger == 0:
            sound = pygame.mixer.Sound("C3B.wav")
            sound.play()

        if finger == 1:
            sound = pygame.mixer.Sound("D3.wav")
            sound.play()

        if finger == 2:
            sound = pygame.mixer.Sound("E3.wav")
            sound.play()

            if finger == 3:
                sound = pygame.mixer.Sound("F3.wav")
                sound.play()

            if finger == 4:
                sound = pygame.mixer.Sound("G3.wav")
                sound.play()

            if finger == 5:
                sound = pygame.mixer.Sound("A3.wav")
                sound.play()

            if finger == 6:
                sound = pygame.mixer.Sound("B3.wav")
                sound.play()

time.sleep(1)
```

Listing 26: wav.py