Imperial College London
Department of Computing

Humanoid Robot Control Using Spiking Neural Networks

Edgars Lazdins

June 21, 2011

Supervised by Dr Andreas Fidjeland and Dr David Gamez

Submitted in part fulfilment of the requirements for the degree of
Master of Engineering in Computing of Imperial College London

Abstract

In recent years there has been an increasing amount of interest in spiking neural networks, which are more biologically realistic than rate based models. Whilst a lot of simulation work has been carried out on spiking neural networks, relatively little research has been done on the use of spiking neural networks to control robots in an environment. The main obstacle to the development of embodied spiking neural networks is the lack of an easy way to link spiking neural network simulators to a robot body in an environment. To address this issue we have developed iSpike: an open source software interface between a humanoid robot, the iCub, and a spiking neuron simulator NeMo. iSpike performs encoding of sensory stimulus received from the iCub into patterns of neuron spikes used by the simulator. The spike output from the simulator is decoded into motor commands for the robot to perform. In our implementation we attempt to mimic the visual, proprioceptive and motor control pathways of the human body. The implementation has been tested with the iCub virtual simulator, the test demonstrates the possibility of real-time control and monitoring of the virtual iCub robot using NeMo. Performance tests show that motor control and proprioceptive pathway performance is efficient enough for real-time application. iSpike is currently available at `http://ispike.sf.net`.

I would like to say thanks to Dr Andreas Fidjeland and Dr David Gamez, who have made the project possible and have helped me throughout, with good advice and valuable feedback.

Contents

List of Figures

1 Introduction

In this report we present iSpike: a bidirectional general purpose interface for use with the NeMo Spiking Neural Network simulator and the iCub robot (Figure 1.1). iSpike was developed to fill in a gap between sensor-enabled robotics and the field of Neural Network simulation. It makes the sensory information gathered by the robot freely available to the Neural Network simulator and allows a robot to be controlled with the use of a spike pattern produced by the Neural Network simulator. iSpike is freely available at `http://ispike.sf.net`.

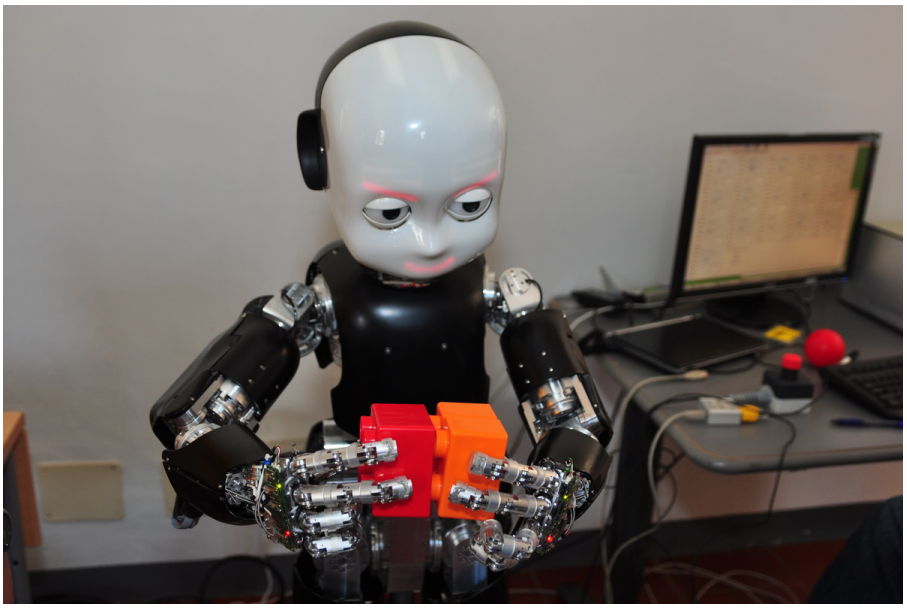


Figure 1.1: The iCub humanoid robot. Image from [26].

The ongoing research in the field of Robotics has resulted in a fair number of very sophisticated robots, including those capable of flying, swimming, and terrestrial locomotion. A common occurrence in most of these robots, is the possession of **rich sensory capability**, including sensors used to gather visual, audio, location, tactile, temperature and even air–flow or water–flow information from the surrounding environment. By gathering the information regarding the surrounding environment, the robot can proceed to create an internal representation of it's surroundings, which can then be used to enable a particular behaviour (such as unassisted flight) or to achieve a goal set for the robot.

Another common feature of many of the recently developed robots, is a relatively **high number of degrees of freedom (DOF)**, which means that the robots quite often have a high number of individual joints, and each joint has a number of individual dimensions of movement, such as vertical/horizontal movement, rotation, compression, twisting and similar. Due to this property, the possible combinations of joints and their dimensions is very large, enabling the robots to exhibit a

complex pattern of movement.

Such robots are interesting to neuroscientists, as they provide an embodied setting and could potentially be used for exploring control and learning, based on neural networks. In particular, the **embodied cognition** field of research "holds that cognitive processes are deeply rooted in the body's interactions with the world". In effect, according to this theory, the organisation of cognition, thoughts and intelligence is heavily impacted, if not defined, by the body the so–called mind is located within. There have been robots developed with this approach in mind, namely the iCub, otherwise known as "an open platform for research in embodied cognition". By using physical robots, researchers can utilise the embodied setting to explore the impact of micro–level behaviours, such as shifts in eye–gaze and position on intelligent behaviour and learning.

Recent research in Computational Neurodynamics has yielded a number of powerful Neural Network Simulators, which are software solutions enabling the design and simulation of a neuron network. In particular solutions, such as SpikeNET and NeMo enable the design and real–time simulation of relatively large ($\sim$100 000 neurons for NeMo) networks of spiking neurons. Spiking neurons are modelled after the biological neurons observed in a variety of biological organisms, including humans, and are characterised by the release of a "spike" in output current in result to a high incoming current. These simulators, coupled with a learning algorithm, such as the Spike Time Dependent Plasticity (STDP) for spiking neurons, enable researchers to explore the behaviour of a given neural network, it's properties and learning capability. This is of special interest, as it is believed, that the basis of biological cognition and, perhaps, consciousness, is the biological brain, which essentially consists of a large, highly inter–connected network of spiking neurons.

By having robots rich in sensory and locomotive capability on one hand, and efficient Neural Network Simulators on the other, it can be seen how it would be of great interest to Embodied Cognition researchers in particular to connect the two together, yielding a system where the information from a robot's variety of sensors is aggregated and channelled to the Neural Network Simulator. This information could be fed into a pre–designed neural network, enabling analysis and transformation. The output of the neural network could then be channelled back to the robot, interpreted as a sequence of motor commands, that the robot performs.

This creates a closed feedback loop, where the robot senses new information, transfers it to the neural network for processing, the neural network then sends a sequence of movement commands to the robot. The movement of the robot results in new information gathered by the sensors. By analogy, if the robot can be assumed to be the part of the human body responsible for carrying out movement and if the Neural Network Simulator is the brain, then the system that ensures a two–way communication between the robot and the Neural Network Simulator can be seen as being similar to the collection of individual body components involved in similar communication between the brain and the rest of the human body.

Ultimately, a bidirectional communication between sophisticated robots and Neuron Network Simulators could lead the way to a new field of potential research and experiments that could further our knowledge on how the biological brain processes sensory information and uses it to achieve goals, and could enable us to create "smarter" robots capable of solving new problems with the aid of neural networks. The goal of iSpike is to realise such connectivity. There currently do not exist any general purpose solutions for enabling information exchange between a robot and a Neural Network Simulator.

2 Background

In this section we provide an overview of the biological pathways we try to emulate with iSpike. Then we proceed to describe the mathematical models that are available to mimic the behaviour of the biological counterparts. We also have a look at the relevant technologies used and their architecture in regards to iSpike. We finally have a brief look at past work we feel was relevant to the development of iSpike.

2.1 Biological Pathways

The human body has a number of biological pathways in place that ensure the communication between the spinal cord and the brain with the rest of the body. These pathways are used to provide the brain with information gathered by sensors of various kind found throughout the body. These sensors have great variation in structure, behaviour and type of information gathered. The methods used to convert the gathered information in a neuron spike pattern representation, that can be used by the brain, also vary greatly.

With iSpike, we aim to mimic the counterpart mechanisms used in the biological human body for each of the pathways we choose to implement. Therefore, what follows now is a brief overview of the mechanisms used by the human body to both gather sensory information and deliver it to the brain as well as the mechanisms used to deliver the motor commands produced by the brain to the recipient muscles or joints.

2.1.1 The Senses

In this section we provide an overview of the methods used by the human body to gather the data acquired by the variety of sensors that exist and deliver it to the brain or the spinal cord in a form of neuron action potentials, otherwise known as spikes. We concentrate on the Visual and Proprioceptive senses as these are the pathways we have included in the initial implementation of iSpike.

Vision

The following description of the human vision system has been paraphrased from [3].

Visual information enters the human eye in the form of electromagnetic waves of varying wavelengths and amplitudes. These waves are refracted by the cornea and the lens and are finally focused on the retina, which is located at the back of the eye. Due to the structure of the eye, a wave approaching from the right hand side would appear on the left hand side of the retina, hence the image on the retina is inverted both vertically and horizontally.

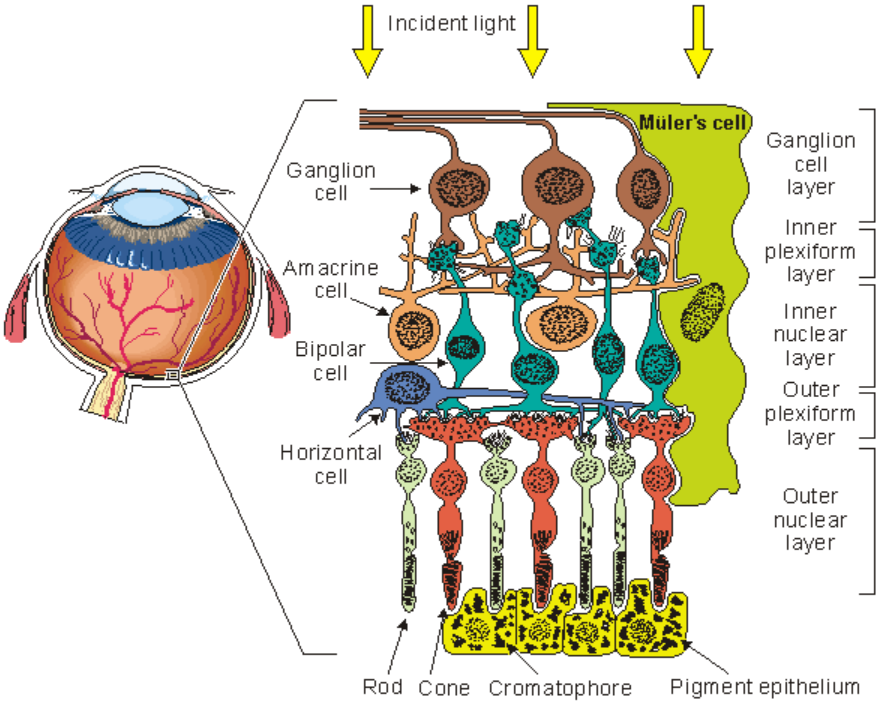


Figure 2.1: Structure of the human retina from [27].

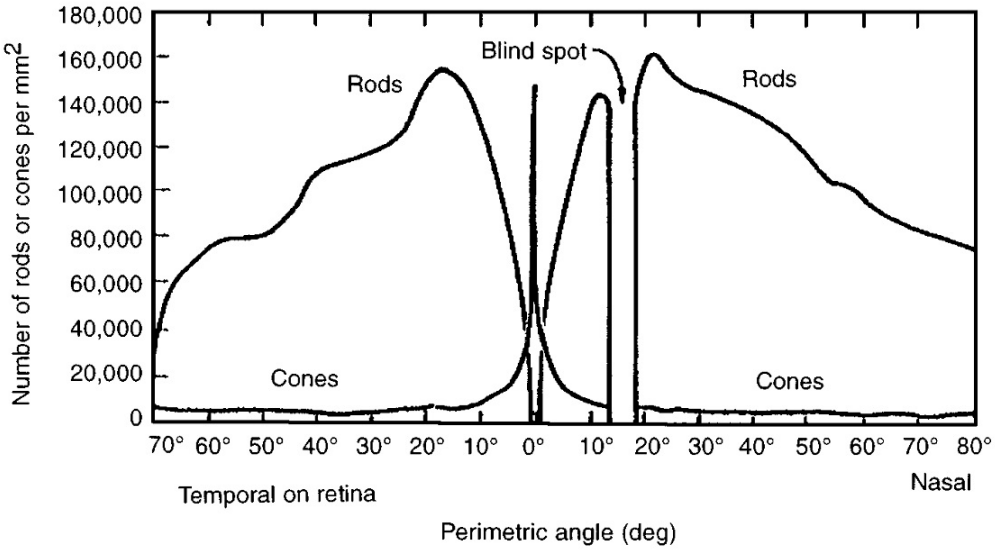


Figure 2.2: Photoreceptor density from [5].

As shown in Figure 2.1, the human retina consists of a number of layers, each containing a different type or types of cells. As light enters the retina, it hits the ganglion cell level, then it travels through a number of layers until finally arriving at a layer of photoreceptors at the very back.

This layer of photoreceptor cells, is the initial starting point of the retinal visual processing. Light can reach these cells with little loss in intensity due to the fact that all the retinal layers above the photoreceptor layer mainly consist of transparent cells. This layer contains two types of photoreceptor

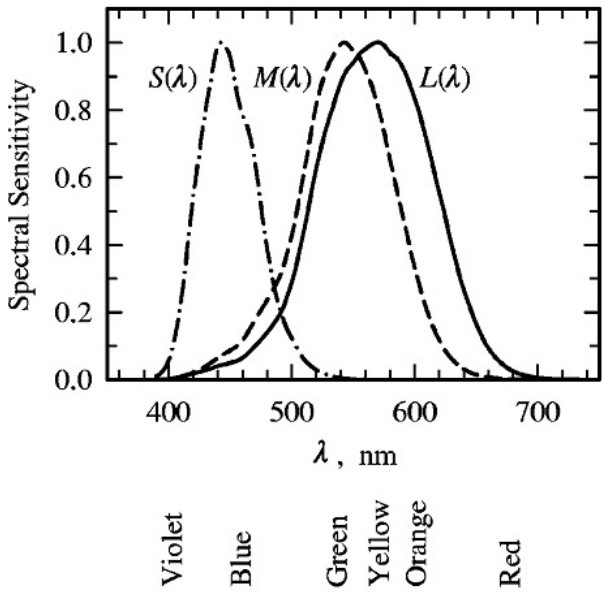


Figure 2.3: Normalised spectral sensitivities of human cones from [35].

cells: the rods and the cones. The name comes from the shape of each photoreceptor.

Figure 2.2 shows the distribution of both types of photoreceptors on the retina. The central or fovea region is almost exclusively covered in cones, whereas the periphery mostly consists of rod photoreceptors.

The rod photoreceptors are over 1000 times more sensitive to light than the cones and during nighttime, it is mostly these photoreceptors that contribute to vision. Conversely, during daytime lighting conditions, it is the cones that are mainly used, as the light intensity is too high for the rod photoreceptors to operate. Due to this difference in sensitivity and the physical layout of the cells on the retina, it is easier for humans to see a star during nighttime by not looking at it directly, but somewhere nearby. Then, the light produced by the star is projected onto the more sensitive rod photoreceptors and star appears brighter. We ignore the presence of rod photoreceptors in our initial implementation of iSpike, as daylight conditions are assumed, throughout.

The cone photoreceptors are colour sensitive and react more intensely in the presence of light of a particular frequency. These photoreceptors are commonly divided in three groups, with each being tuned to a different frequency. The S group cones are tuned to light with a wavelength of 430 nm, which corresponds to the red colour. The M group cones respond mostly to 530 nm light, which corresponds to the green colour and finally the L group photoreceptors mostly respond to 560 nm light, which corresponds to the yellow colour.

As the light hits a photoreceptor, a number of neurotransmitters is released in proportion to the intensity and wavelength of the incoming light. These neurotransmitters are used to deliver information to the next layer of retinal cells located directly above the photoreceptors.

The next layer of cells relevant to the visual pathway are the bipolar cells. Bipolar cells are either in direct contact with photoreceptors or in an indirect contact via another cell type, named the horizontal cells.

The Bipolar cells are divided in two types: The ON bipolar cells respond more strongly to the presence of light and the OFF bipolar cells respond to the lack of lighting. Each bipolar cell received input from a cluster of photoreceptor cells. The main function of these cells is to convert the neurotransmitter signal received from the photoreceptors into an electric current signal used by the cells in the next layer. The output of electrical signal makes these cells similar to the neurons found in the brain, the difference is that the output from the bipolar cells is analogue and the intensity of the signal is proportional or inversely proportional to the input, depending on the cell type. As will be shown later, the output of the neuron cells in the brain, instead, takes the form of action potentials and as such, is digital as opposed to analogue.

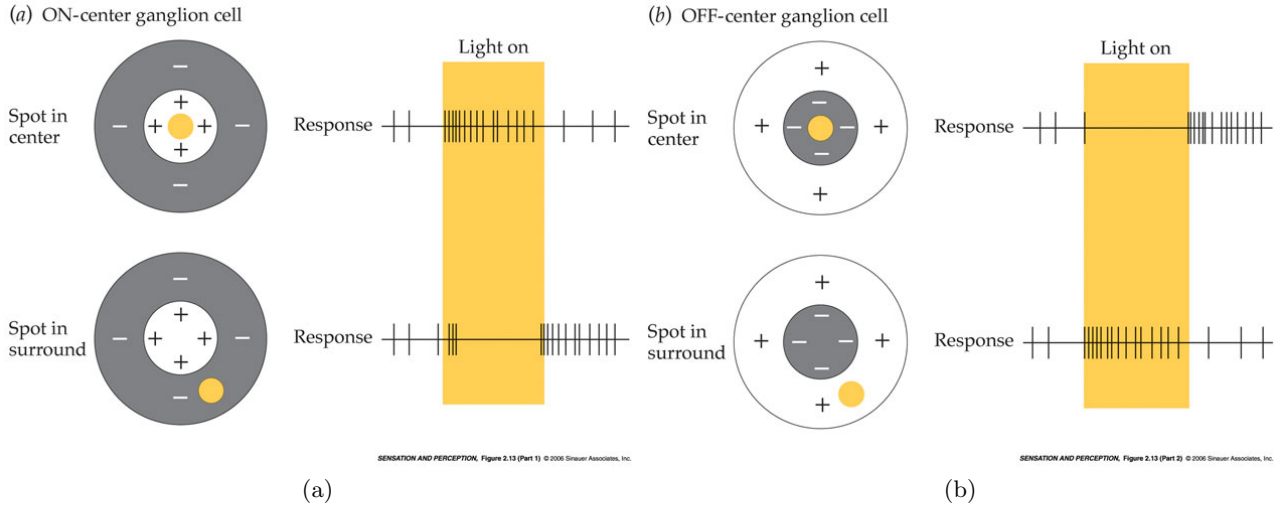


Figure 2.4: Ganglion cell receptive fields from [33].

The next retinal layer consists of ganglion cells, which receive the electrical signal produced by the bipolar cells. Each ganglion cell receives it's input from precisely one bipolar cell. Each ganglion cell has a *receptive field*, which is the region in the original image that impacts the output of the cell. The receptive fields are circular in shape and are divided into two regions: the center and the surround. These cells are also divided into two types: The ON-centre ganglion cells respond to presence of light at the center of the receptive field and lack of light in the surround. The OFF-centre cells are opposite in behaviour and will respond to light in the surrounding area of the receptive field and lack of lighting at the center. Both types of ganglion cells do not respond at all if all of the receptive field is equally illuminated. The output of the ganglion cells is digital and takes the form of action potentials, which are then transmitted to the lateral geniculate nucleus (LGN) part of the brain.

The ganglion cell layer actually consists of three types of cells, each differing slightly from the others. The Magnocellular cells, covering approximately 5% of the layer, have large receptive fields and respond to stimulation with a transient burst of action potentials. The parvocellular cells, covering approximately 90% of the ganglion cell layer, have smaller receptive fields and if stimulated, produce a steady stream of action potentials as long as the stimulus is present. Some parvocellular cells are colour opponent and produce a stronger signal if light of one colour is present in the central area of the receptive field and another colour is present in the surround. There are three types of colour

opponency. Red+Green-, Green+Red- and Blue+Yellow-. The first part of the name is the colour present at the center of the receptive field and the second part is the colour present in the surround, for example, a Red+Green- parvocellular cell will respond to the presence of red colour in the central region of the receptive field and green colour in the surround.

Proprioception

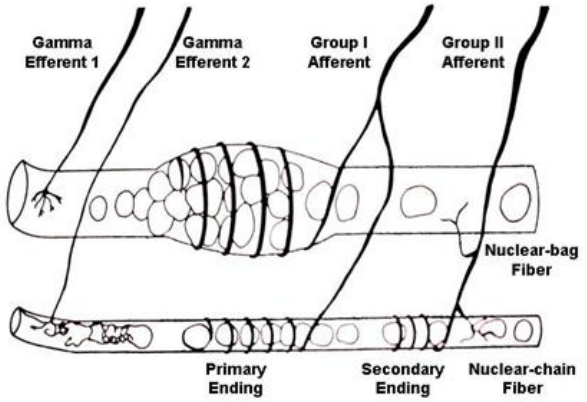


Figure 2.5: A diagram of a muscle spindle. From [7].

Proprioception is the somatic sensory system that is responsible for the identification of body part ownership and also providing information about the position and movement of individual body parts. Most of the muscles in the human body contain specialised structures, called *muscle spindles*, otherwise known as *stretch receptors*. A stretch receptor consists of several types of specialised muscle fibres enabling the detection of changes in the muscle length, which is directly related to the joint position. Figure 2.5 contains a diagram of a muscle spindle.

Muscle tension is detected by another type of sensors commonly found in muscles: the *Golgi tendon organ*. This sensor acts like a strain gauge and monitors the tension in the muscle or the force of contraction.

The output from these two types of sensors is delivered to the motor control regions of the spinal cord and via spinal cord to the cerebellum region of the brain, which is also primarily responsible for motor control.

2.1.2 The Brain

The Biological Neuron

"Neurons are the principal cellular elements that underlie the function of the nervous system." [23] A biological neuron receives incoming electrical current via a network of dendrites, which normally extend no more than 2 mm in length. The electrical current is delivered to the neuron body, also called the soma. From the soma, the electrical current travels through (usually one) output "channel" called the axon. Axons are microscopically thin, but can be relatively long and carry a signal to different regions of the brain, connecting with as many as 10000 other neurons.

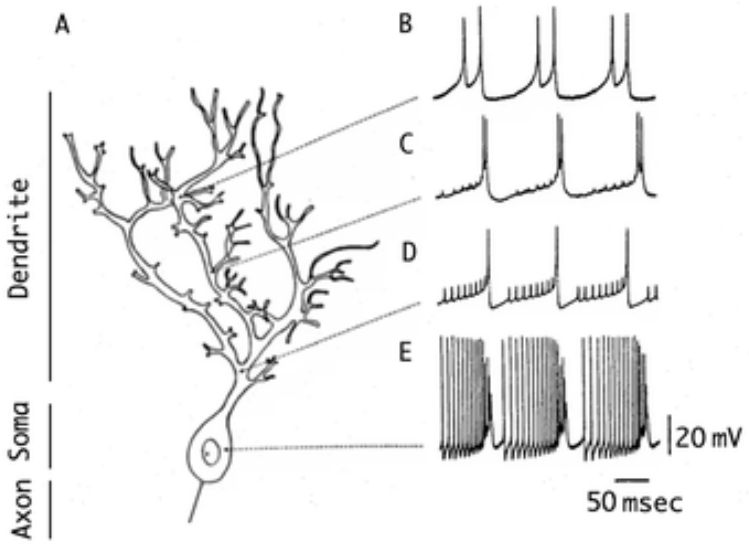


Figure 2.6: A picture illustrating the appearance of a single neuron along with spike pattern recordings from various parts of the neuron. [23].

The region where an axon comes in close proximity to a dendrite of another neuron is called the synapse. Axons do not come into physical contact with dendrites, instead, there is a small gap where complex electrochemical process enables the transfer of electrical current from the axon to the dendrite.

Information is transported via neurons in the form of spikes, or short electrical pulses with increases current. Spikes travel toward the body of the neuron along it's dendrites, then an electrical charge builds up in the body of the neuron. Once the electrical charge reaches a critical value, an outgoing spike is released along the axon and the electrical charge built up in the soma decreases.

Vision

Lateral geniculate nucleus The spikes produced by the Ganglion cells are delivered to the Lateral geniculate nucleus part of the brain via the optic nerve. There are actually two lateral geniculate nuclei in the human brain, with one located in the left hemisphere and one in the right. The signal from the left eye is passed to the LGN in the right hemisphere and vice versa. This happens in the region where the optic nerves coming from each eye cross, otherwise known as the *optic chasm*. The LGN in structure is organised in layers, sorted by the type of the ganglion cells that provide input to each region. The function that the LGN itself performs in visual perception is currently unknown.

Primary visual cortex The majority of the outgoing connections from each LGN end up in the Primary visual cortex area of the brain, which is located in the central area of the back of the brain. Most of the neurons in the primary visual cortex are orientation or direction sensitive, this implies that this region of the brain performs at least some kind of pattern recognition and movement analysis tasks.

2.1.3 Motor Control

Motor Control can abstractly be divided in two independent components:

- Individual muscle contractions, resulting in joint movement. These are controlled by the spinal cord in the human body.
- Individual "motor programs" stored in the spinal cord. These are controlled and executed by the human brain.

The brain-stem and the spinal cord contain a toolbox of motor programs, these are neural networks enabling the execution of sequential muscle contractions, leading to the execution of a simplistic behaviour, for example locomotion, eye movement, breathing and similar [17]. The brain is not concerned with the order and nature of component commands involved in, for example, raising one's arm. Instead, the brain would send a trigger to the spinal cord that indicates that this movement should be carried out, and the spinal cord then executes the individual actions required to achieve the goal of the program. A good demonstration of this phenomenon, is based on the observation that a chicken can exhibit the complex behaviour of running, even with it's head removed from the rest of the body. The human brain initiates a particular motor program with the help of axons that originate in the brain and descend through the spinal cord along two major groups of pathways.

2.2 Mathematical Models

2.2.1 The Senses

Models for Vision

We will now briefly look at what models there exist for simulating the individual components involved in human vision. As we've seen before, the core component of vision is the retina. Modelling the human retina can be notionally divided into the following two levels:

- Photoreceptor level
- Ganglion cell level

At each level a different model can be used to simulate the information transformation that occurs in the biological retina.

Photoreceptor Level

The most important feature of the photoreceptor level, is that their distribution is nonlinear, as there are more photoreceptors near the center of the retina. They are also laid out in a circular fashion and have receptive fields that can overlap. The size of the receptive fields increases closer to the edge of the retina. The overall effect achieved is that of **foveation**, and we will now look at some models that try to achieve this. The $log(z)$ and the $log(z + a)$ models do not account for overlapping receptive fields, whereas the Wilson's Model does. The following model descriptions have largely been paraphrased from [5].

Most of the models here assume transforming the original image from *cartesian* to *polar* coordinates. Figure 2.7 shows side by side the two coordinate systems. In Cartesian coordinate system a point is identified by the distance from the origin in every dimension, in polar coordinates a point is identified by the 1–dimensional distance and angular distance from the origin. An image can be transformed from Cartesian to polar coordinates by first creating an empty two–dimensional image in the polar plane and then iterating over each point in the polar plane and sampling the pixel value from the Cartesian coordinates. The conversion from polar to Cartesian coordinates is similar.

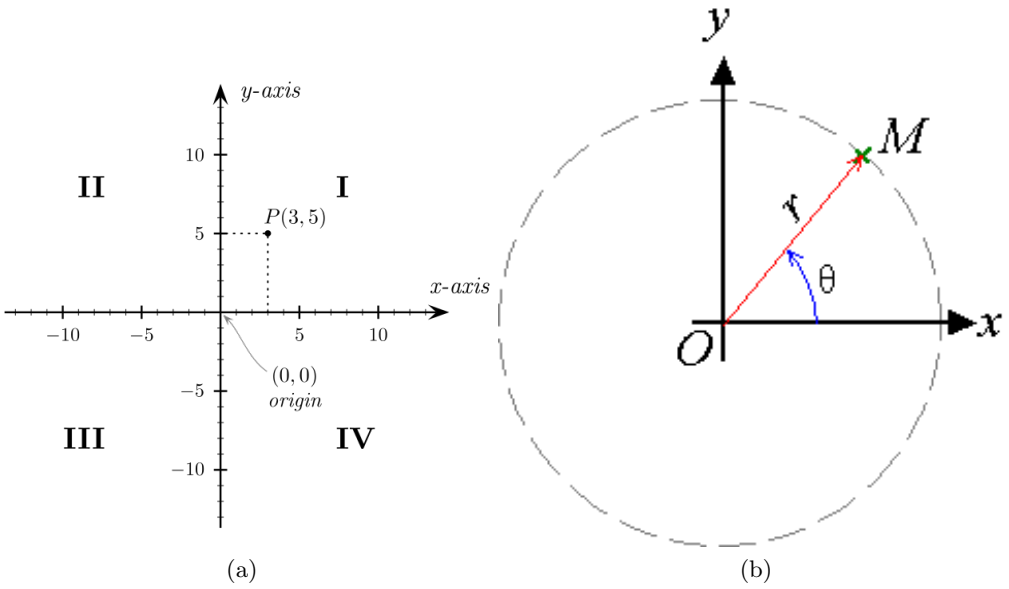


Figure 2.7: Cartesian (left) and Polar (right) coordinate systems. Images from [33].

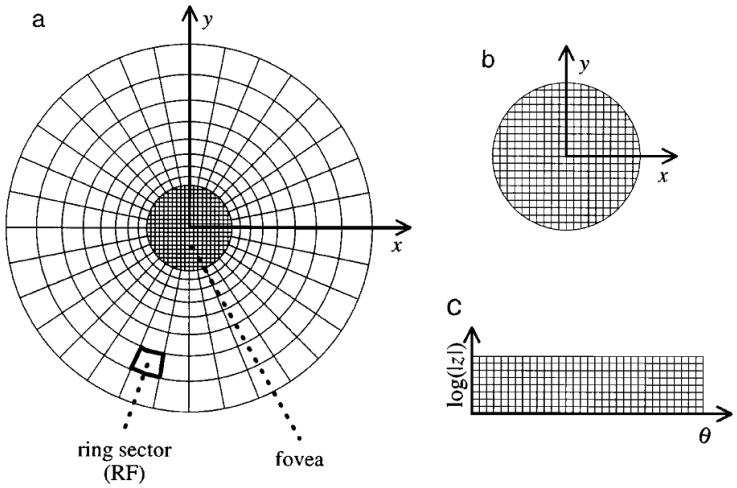


Figure 2.8: The log(z) model. (a) mapping template, note the separation of central and peripheral regions. (b) The central output grid. (c) The peripheral output grid. Image from [5].

The log(z) Model In this model, the number of photoreceptors decreases exponentially with the distance from the center, so that the resultant pixel x is responsible for all pixels z in the original image, such that $log(z) = x$. x and z are the distances from the center. In order for the resultant image to be circular, these transformation are done in the polar coordinate system, hence this approach is often called the **log-polar transform**. It's advantage is in the simplicity of the transformation, enabling high efficiency. A disadvantage is that the pixels near the centre of the image will have a negative log value. To alleviate this, the central area is normally sampled at the original resolution and the transform is only applied to the periphery.

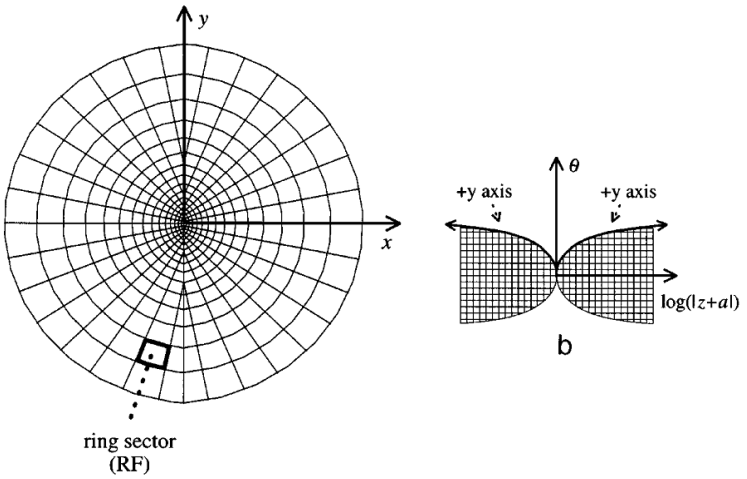


Figure 2.9: The log(z+a) model. Left hand side shows the input mapping template, right hand side demonstrates the output grid. Image from [5].

The log(z+a) Model It has been argued that the $log(z)$ model is inadequate for the retinal image, due to the singularity of the logarithm at 0. The $log(z + a)$ model tries to get past this issue by using $log(z + a)$ as a mapping function, with a hand picked value for the constant a. To perform the mapping, the image is divided into two half-planes along the vertical mid-line, the right hand side is mapped using the equation $log(z + a) = x$ and the left hand side using a complementary equation $2log(a) - log(-z + a)$. This produces a *butterfly* like image, where the coordinate system is log–polar, but with exchanged axes. When mapped back to the Cartesian coordinates, this model does not produce an empty ellipse at the centre and hence accounts for the photoreceptors in the vicinity of the central retina more accurately.

Wilson's Model This model was proposed by Wilson [38] and it involves creating concentric rings, representing the receptive fields on the image. The rings overlap by a given magnitude (parameter of the model) and increase in size with distance from the centre (parameter of the model). This model, much like the $log(z)$ model, does not account for the central area of the retina, and hence, a uniform distribution of receptive fields needs to be assumed there. This model is more biologically plausible due to two reasons: firstly, the receptive fields are circular instead of being rectangular as in the previous models. Evidence suggests, that the receptive fields of photoreceptors in the biological

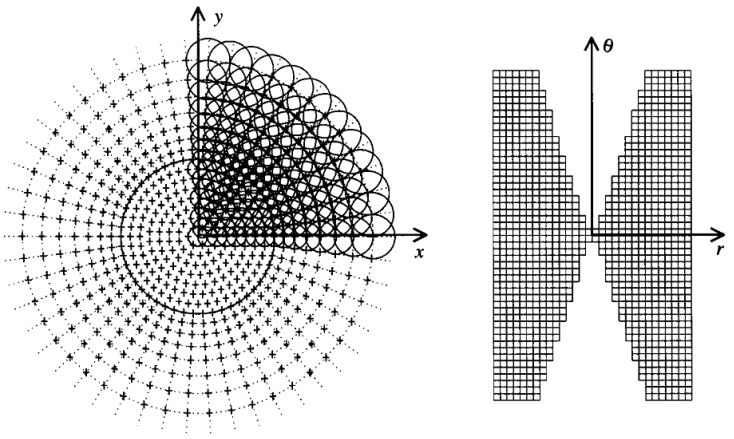


Figure 2.10: Wilson's overlapping circular RF model. Input mapping template on the left, output grid on the right. Crosses on the left hand side indicate the receptive field centres. Image from [5].

retina are circular. Secondly, The receptive fields overlap, as they do in the biological retina. This model is less computationally efficient due to the fact that a single pixel in the original image could contribute to multiple receptive fields, due to them overlapping. One application of the Wilson model is the publication by Yamamoto et al. [39], who describe an implementation of a foveated robot vision system that simulates non-linear sampling using an adaptation of the Wilson model. They further design an Integrative Iconic Memory system to integrate the original foveated snapshots to create one encompassing representation of the surroundings. Together with a generated saliency map, an active vision system Fovea is created, capable of shifting the vision to points of interest in the surroundings.

Ganglion Cell Level

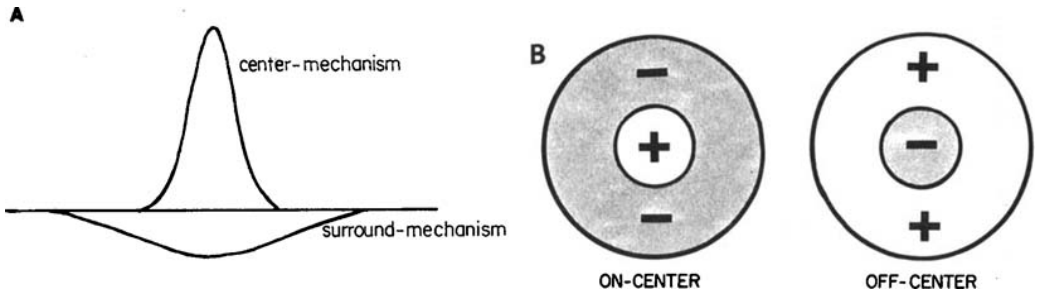


Figure 2.11: This image illustrates the principle of using Difference-Of-Gaussians filter to approximate the behaviour of a ganglion colour opponent cell. Image from [18].

The most popular method for simulating the on–centre and off–centre Ganglion cells is the **difference-of-gaussians**. Originally devised by Enroth–Cugell et al. in 1966 [9], it involves, for example in the case of the R+G- colour-opponent cells, first creating the Red and Green versions of the original image. Then a Gaussian filter is applied to each image with different standard deviations σ yielding a Red+ and a Green- image. The final R+G- image is obtained by subtracting the Green- image

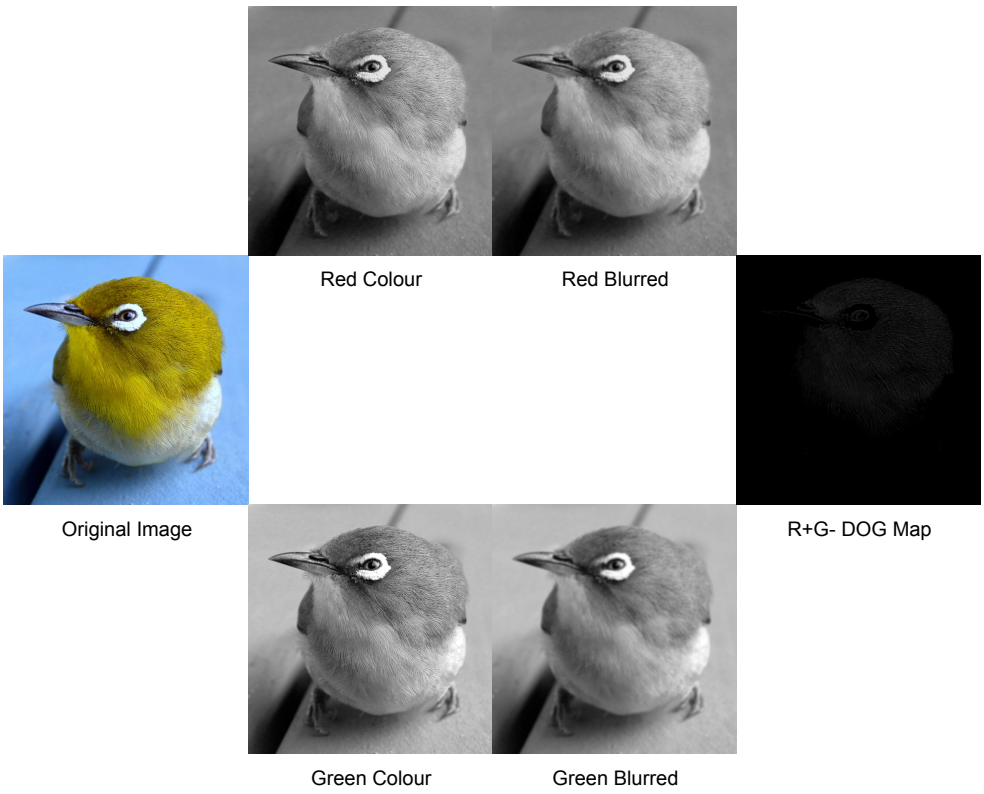


Figure 2.12: The process of generating a single opponency map using the Difference of Gaussians method.

from the Red+. This process is shown visually in Figure 2.12 This model approximates the colour opponent cells due to the nature of the Gaussian convolutions. By applying a Gaussian convolution to an image, a Gaussian curve is fitted on top of each pixel in turn and the updated pixel intensity is the weighted average of the surrounding pixels, hence in the case of the Red+ image, a pixel will have a higher intensity, if there are high intensity pixels near it, much like a receptive field that is positively influenced by the presence of red colour near the center would release a stronger signal. By subtracting the G- image from the R+ image, we effectively add the feature of being negatively impacted by green colour in periphery to the pixels found in the R+ image. Hence, if each pixel is an approximation of a receptive field, the output value will be the sum of positive influence from red colour near the center and negative influence from the presence of green colour near the periphery. Image 2.11 illustrates this principle. Figure A shows two Gaussian functions, one positive, one negative with varying intensities. Figure B shows an on–centre and an off–centre Ganglion cell receptive field.

Orabona et al. [31] describe a model for object–based visual attention and implement this model in the iCub robot. They sequentially apply a $Log(z)$ foveation algorithm, a Difference of Gaussians transform and an Edge detection algorithm to yield a saliency map from the input image. This saliency map is then used to guide the gaze of the robot in search for interest areas in the surroundings.

2.2.2 The Brain

Mathematical Neuron Models

A number of models have been developed to approximate the behaviour of a biological neuron.

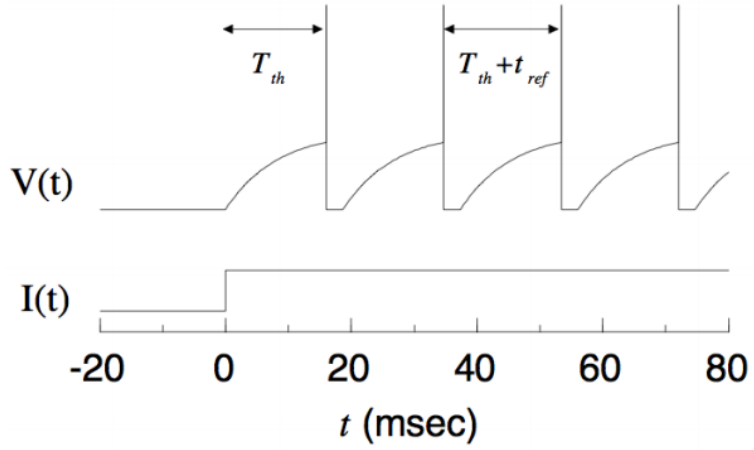


Figure 2.13: Voltage dynamics of a single Integrate and Fire neuron, subject to a constant input current. Image from [21].

The Integrate and Fire Neuron Model One of the simpler models used, is the **Integrate and Fire** model, where much of the biological detail of the neuron behaviour is abstracted away, such as the refractory period, where once a neuron releases a spike, it is more "difficult" to release a subsequent spike, and a higher incoming current is required. The equation at the core of this model is as follows:

$$\tau_m \frac{dv}{dt} = -v(t) + RI(t)$$

Equation adapted from [12]. v is the membrane potential, τ is a time constant and I is the incoming current at time t. This model abstracts away from spike time dynamics and instead, when membrane potential reaches a critical value, it is set to an instantaneous value representing the occurrence of a spike (otherwise known as a Dirac pulse) and then reset to a resting value.

While this model is very computationally efficient and therefore allows the simulation of larger and more complex neural networks, it ignores many important aspects from the behaviour of actual biological neurons, making it a relatively poor choice when used in a simulation of any biological relevance.

The Hodgkin-Huxley Neuron Model The Hodgkin-Huxley neuronal model tries to capture the dynamics of the chemical events that occur within a neuron to a great detail, originally published in 1952 by Huxley et al. [19]. It is governed by the following equation:

$$C_M \frac{dV}{dt} = I - I_i$$

where

$$I_i = I_{\mathrm{Na}} + I_{\mathrm{K}} + I_{\mathrm{l}}$$

I	is the total membrane current density
V	is the displacement of membrane potential from its resting value
C_M	is the membrane capacity per unit area (assumed constant)
t	is time
I_{Na}	is the current carried by sodium ions
I_{K}	is the current carried by potassium ions
I_{l}	is the current carried by other ions

The behaviour of neurons simulated according to this model closely mimics the behaviour of biological neurons, including a refractory period and the support for a variety of neuron types, each exhibiting a unique behaviour. A major disadvantage of this model, is that it is computationally difficult and hence it is not possible to use it for larger neuron networks.

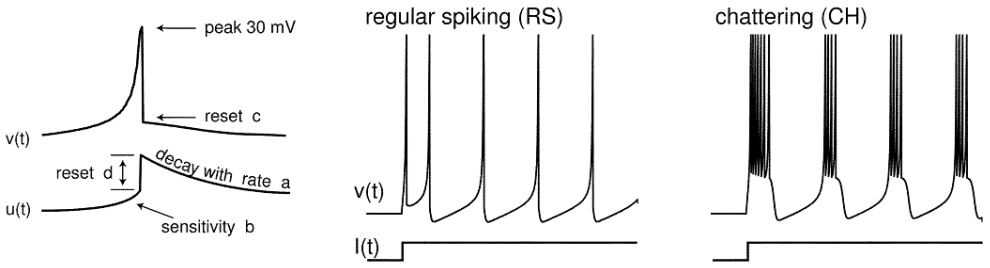


Figure 2.14: Visual portrayal of the Izhikevich Neuron Model parameters and two sample neuron behaviours that can be generated by the model. Image adapted from [24].

The Izhikevich Neuron Model The Izhikevich neuron model has been developed relatively recently (2003 publication by Izhikevich [24]) and it aims to combine the biological plausibility of the Hodgkin-Huxley model with the computational efficiency of the Integrate-And-Fire neurons. The core equations of the model are as follows:

$$\begin{aligned} \frac{dv}{dt} &= 0.04v^2 + 5v + 140 - u + I \\ \frac{du}{dt} &= a(bv - u) \end{aligned}$$

where

v	is the membrane potential
u	is a membrane recovery variable
a, b, c, d	are model parameters

If the membrane potential v reaches a value of 30 mV, it is reset to a resting potential c and the value of the recovery variable u is increased by d. The intuitive meaning of the recovery variable u, is that higher values of u slow down the rate of increase of v, making it more difficult for the neuron to fire. Such behaviour enables the Izhikevich neurons to have a refractory period. By adjusting the values of the four model parameters, a variety of neurons with different behaviours can be produced. The Izhikevich neuron model is computationally efficient enough for relatively large scale simulations, while preserving biological plausibility to a high degree.

Neural Coding

In the previous chapters we have shortly outlined the biological pathways from sensory and locomotive regions of the body to the brain and vice versa. We have also looked at the behaviour of individual neurons. Another topic we need to look at, is that of Neural Coding. Neural Coding is a neuroscience related field that tries to understand the actual encoding and decoding methods used by the body to transform analogous sensory information into a digital spike pattern or to transform a digital spike pattern into an analogous command for the body to perform. It is a very active research field and there is much debate as to the actual methods used by biological organisms in different creatures and body parts. The following has been largely paraphrased from the "Pulsed Neural Networks" book by Maass et al. [25]

Rate Coding One possible coding method used by is that based on the firing rate of the neuron. In this approach, the rate at which a neuron fires is proportional or inversely proportional to the intensity of the underlying stimulus. The rate is interpreted as either the average firing rate over time, average firing rate over several repetitions of an experiment, or average over a population of neurons. For example. there is evidence of Rate Based coding used in the stretch receptor in a muscle spindle [2] as well as in the auditory cortex of marmoset monkeys [4]. A common critique of rate based encoding schemes is that a considerable amount of time is required to obtain an accurate value for the mean firing rate, whereas there are many situations where organisms respond to changes in an environment more quickly than an accurate value could possibly be acquired, for example face selective neurons can respond only 80-100 ms after stimulus onset [32]. Another criticism for this method of encoding is that it can lead to high inefficiencies in terms of the number of neurons required to encode a particular phenomenon. It is commonly acknowledged that a single neuron cannot accurately encode a stimulus by using a rate based scheme, in fact, according to Gautrais et al. [14] to encode a single analogue value to an accuracy of ± 10 Hz would need no less than 281 independent neurons.

Temporal Coding Another approach to encoding analogue signal into neuron spikes, is one based on the temporal difference between individual spikes. In this approach, the intensity of the stimulus is related to the absolute latency of the first spike to arrive, or to the relative latency between the first spikes to arrive at a particular neuron. For example Gollisch et al. [16] were able to accurately reconstruct the image displayed to a salamander retina by using a relative latency encoding scheme. A major advantage to this method, is that a small number of spikes is sufficient to accurately derive the intensity of the original stimulus, hence it could account for low latency times of reaction to

environmental stimulus. A disadvantage of this approach, is that it is not immune to random noise in the form of nondeterministic variation in the latencies of individual spikes, which could affect the decoded value (otherwise known as noisy temporal jitter).

Rank Order Coding Yet another popular candidate for an encoding scheme is based on the order of arrival of the first spike from the incoming neurons. In this approach, we look at the order of neurons from which we receive a spike. It is assumed that each ordering encodes a possible value related to the original stimulus. Hence, for example, 6 neurons can potentially encode $6! = 720$ distinct values, as there are 720 possible ordering. Each possible ordering is assigned a rank or intensity value. An advantage of this approach is that it is immune to noisy temporal jitter, it is invariant to changes in contrast and luminance, if used to represent images, and it can be used to encode large amount of information very rapidly [14]. There is no clear evidence of rank based encoding use in a biological body, but it has been used successfully to encode and decode images [37].

Population Coding A final method of neural coding we look at, is *population coding*. This method implies that the original source of the signal is not encoded by a single neuron value, but instead, a population of neurons is used, such that the collective activity is related to the intensity of the sensory stimulus. A value for the collective activity (also known as the population vector) is retrieved by calculating the mean of the neuronal activity of a given population of neurons. In an experiment performed by Georgopoulos et al. [15], where a monkey was trained to move a joystick in a number of directions, it was shown that the direction of movement in the arm is strongly correlated to the population vector value received from the relevant neuron groups in the primary visual cortex.

2.2.3 Motor Control

Spinal Cord Level At a spinal cord level, the main component in terms of motor control, is the Central Pattern Generator (CPG). Due to the fact that pattern generating networks commonly produce cyclic patterns of muscle activity, they are commonly modelled as oscillators, or groups of coupled oscillators. For example Figure 2.15 shows a model used by Ijspeert et al. [20] in a salamander robot, capable of swimming and walking on terrain. The model consists of interconnected custom oscillators.

An oscillator, mathematically, is a function that depends on time and shows rhythmic repetition in it's value with respect to time. An important property seen in the central pattern generator, is the capability of oscillating network groups to synchronise, so as to produce unified response to stimulus.

One commonly used oscillator model is the *Kuramoto model*. Kuramoto [22] provides a mathematical model for a population of N coupled phase oscillators $\Theta_i t$ having natural frequencies ω_i distributed with a given probability density $g(\omega)$. These variable are related by the following equation:

$$\frac{d\theta_i}{dt} = \omega_i + \sum_{j=1}^{N} K_{ij} \sin(\theta_j - \theta_i) \qquad i = 1, ..., N$$

Here, K_{ij} is a coupling matrix that defines the level of coupling between each pair of oscillators in the population. If the degree of coupling is low, each oscillator will run independently at it's natural

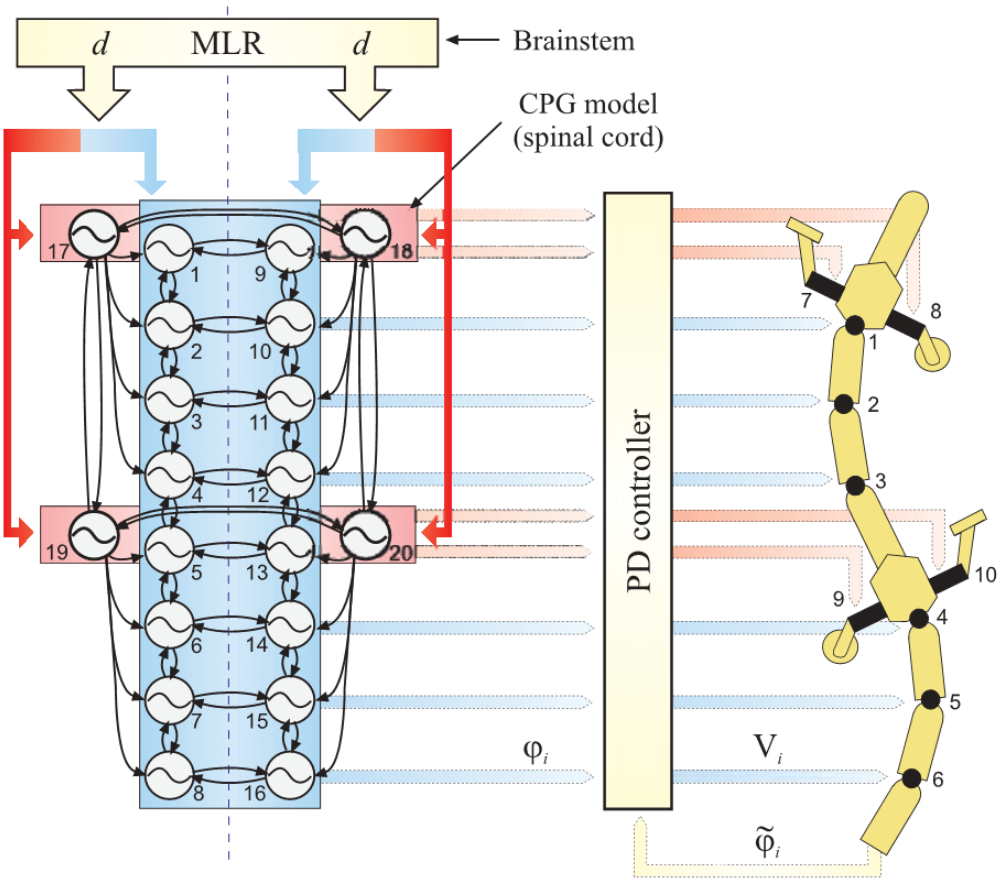


Figure 2.15: A CPG model using custom oscillators used to control a salamander robot by Ijspeert et al. [20]

frequency. If the coupling is sufficiently high, collective synchronisation emerges spontaneously. In their publication [8], Cruz et al. describe a computer simulation of a CPG that relies on 1000 coupled Kuramoto model oscillators distributed in a 10 x 10 lattice. The synchronous oscillatory activity given by the simulation was of the same order of magnitude as the one observed within the sensorimotor cortex of monkeys and humans.

Brain Level Most of the higher level motor control in the brain occurs at the Primary Motor Cortex (PMC). There is evidence that a population based encoding is used in this detail as described by Georgopoulos et al. [15] in their experiment. In this experiment, the authors showed that the neurons in the PMC are highly tuned to a particular movement direction and respond with higher intensity during movements that occur in this preferred direction. It follows that using a set of artificial neurons, with each having a preferred direction could be used to extract the direction of overall movement that an incoming pattern of neuron spikes represents.

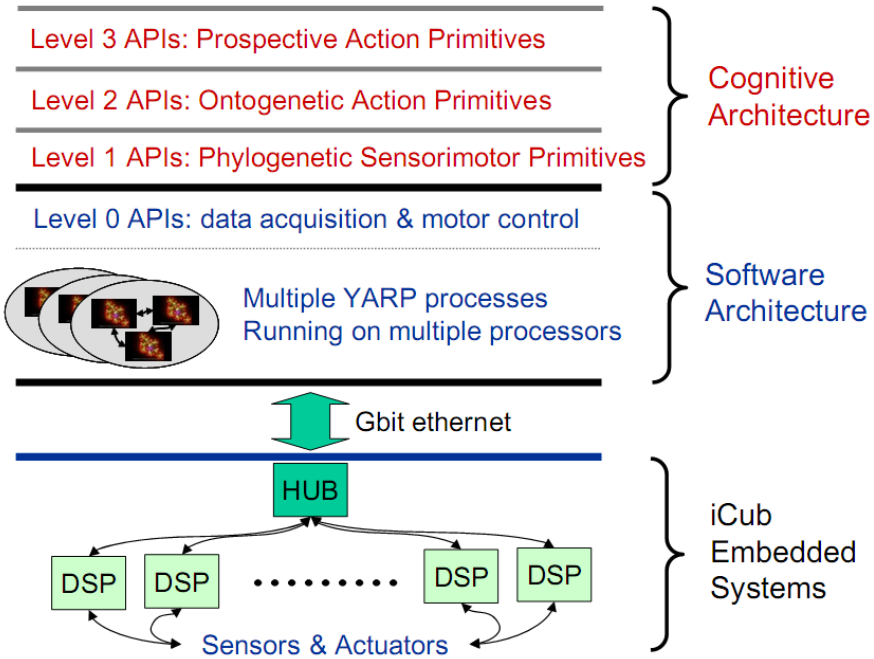


Figure 2.16: The layers of the iCub architecture. Image from [28].

2.3 Technology Used

2.3.1 iCub

For the purposes of this project we used a humanoid robot named iCub. A picture of the iCub is available in Figure 1.1. iCub was developed as a part of the RobotCub collaborative project funded by the European Commission under the sixth framework programme (FP6) by Unit E5: Cognitive Systems, Interactions and Robotics.[29] The iCub aims to replicate a 2 years old child both in appearance and capability. To achieve this, iCub is equipped with a rich variety of sensors, including "digital cameras, gyroscopes and accelerometers, microphones, and force/torque sensors"[29] as well as 53 degrees-of-freedom, enabling complex locomotion and behavioural patterns. The iCub is also a freely-available open system, all of the mechanical an electronic design has been released under a GNU Free Document License (FDL) and all of the software is released under a GNU General Public License.

iCub utilises YARP (Yet Another Robot Platform) for all communication and higher level information processing. Figure 2.16 demonstrates the levels of the iCub architecture.

YARP is a set of libraries that support modularity by abstracting two common difficulties in robotics: namely, modularity in algorithms and in interfacing with the hardware.[29] Some of the core features of YARP that differentiate this platform it's competitors are as follows:

- *Distributed control:* YARP naturally decouples the processes used to control a robot from the robot itself. A large number of independent computers could be involved in processing sensory data from a robot and issuing motor commands in return. This allows the use of higher processing

power to control the robot, which, in turn, allows controlling more sophisticated robots and processing of higher quality sensory information in real-time.

- *Modularity:* The architecture of YARP itself and the YARP processes used to control a robot are highly modular. For example, a YARP process can be independent of location and can be moved across machines at runtime. This enables efficient recovery from hardware failure.

- *Minimal interference:* Due to the fact that, often, the performance of a robot controller depends on the timing of various signals, YARP is designed in such a way, that the addition of a new component has close to no effect on existing processes, provided enough computational resources are available.

- *Resilience:* YARP does it's best to reduce dependencies between individual processes. Communication channels between processes can be created and destroyed without affecting the processes themselves. If a process terminates unexpectedly, the process it communicates to does not have to be restarted. This increases the overall resilience of the system against runtime issues and failure. A single component can be restarted or terminated without affecting the rest of the system.

- *Compatibility:* The YARP architecture is created with an open world in mind and it is not tied to a particular system environment or conditions. It is compatible with a multitude of operating systems (Windows, Linux, Mac OSX, QNX6) and relies on the ACE library for connectivity and concurrent functionality, the ACE library is platform independent as well. This makes YARP much easier to adapt to a given environment and makes it easier to use in a collaborative environment with a variety machines with different operating systems working on the same project.

Communication in YARP occurs via *ports*. Ports are one directional channels that allow the transfer of information in a particular image, for example text, images or sound. A client can connect to a YARP port directly, or they can be connected to each other, forming a network. For example, figure 2.17 demonstrates a network of YARP ports. Ports belong to *processes* as demonstrated in the figure, for example, the port */camera* belongs to the process *yarpdev*. YARP processes can be distributed among a number of machines, each machine could be running a different operating system. A variety of transport protocols are available for use with YARP, including TCP, UDP and multi-cast. Each protocol is more suited for given application. YARP ports can be created, connected with each other and destroyed at runtime.

At the core of the YARP framework, there is a *nameserver*. A nameserver is a special port that becomes available as soon as a YARP server is started on a machine. The nameserver acts as a *directory of ports*. A client can use the nameserver to find out the IP addresses and system ports of a particular YARP port. For comparison, much like a DNS nameserver enables the resolution of a domain name to an IP address, so does a YARP nameserver enable the resolution between a YARP port name and the IP address and system port of that port name.

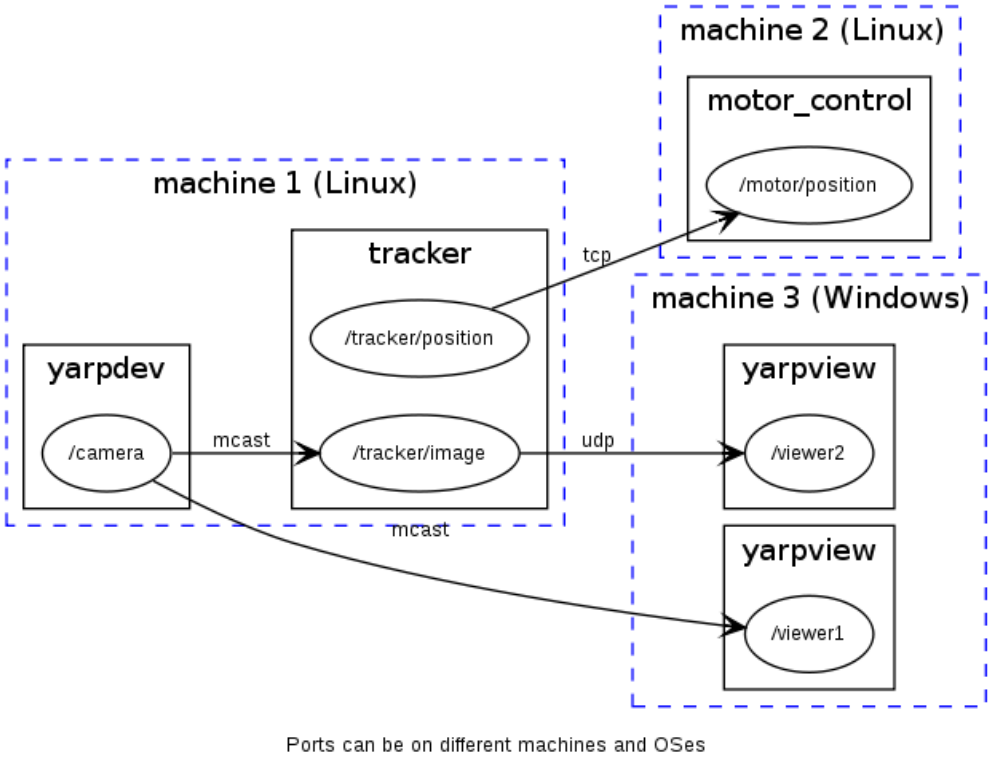


Figure 2.17: An example of a network of YARP ports. Image from [1].

2.3.2 NeMo

NeMo is a Spiking Neural Network (SNN) simulator, that enables high performance real-time simulation of around 40000 Izhikevich neurons delivering up to 400 million spikes per second with the use of highly parallel commodity CUDA enabled graphics processing units (GPUs). For comparison, an adult brain has approximately 10^{11} neurons and approximately 10^{14} connections, so faithfully simulating even a subset of the human brain is currently not within our reach. NeMo is a C++ class library with APIs for Python, PyNN, Matlab, and pure C. Learning is supported through spike-timing dependant plasticity (STDP). [10] Figure 2.19 demonstrates performance metrics of the simulator. The left figure shows the speedup measured with respect to real time performance for a network of 30,000 neurons with a variable number of synapses per neuron, running on a C2070 GPU. The right figure shows throughput for the same network and hardware, measured in terms of spikes delivered per second.

2.3.3 SpikeStream

SpikeStream provides a graphical user interface to the NeMo simulator. It provides a visual way to create, manage and simulate networks of spiking neurons. Spike Time Dependent Plasticity (STDP) learning is also supported. During simulation, SpikeStream presents the user with real-time information on changes in connection weights as well as occurrences of neuron spikes. Spike patterns can be injected in a predefined network at runtime. SpikeStream provides a variety of analysis plug-ins, such as testing for Liveness and performing State-based Phi analysis. A MySQL database is used

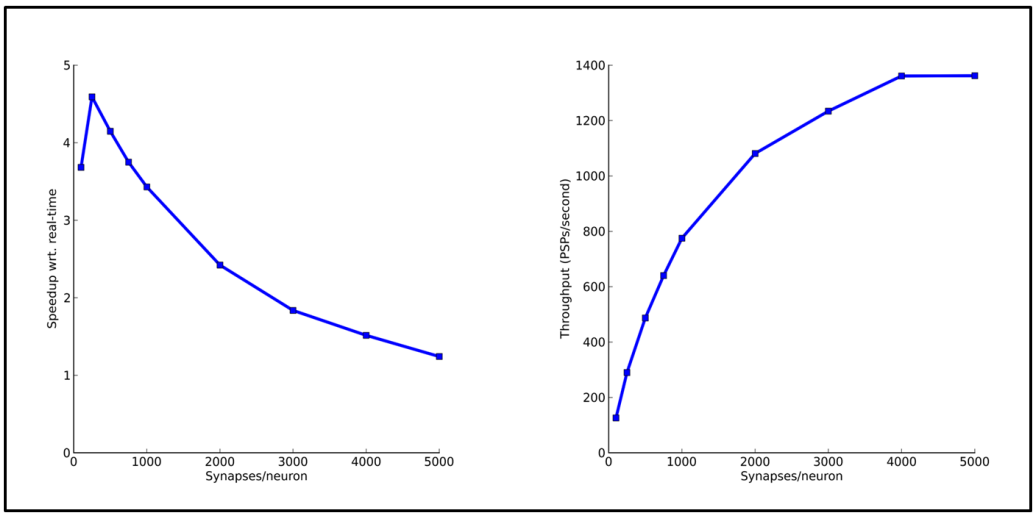


Figure 2.18: NeMo performance metrics. Image from [1].

as the back-end for storing neurons, connections, parameters, archives and analyses. Key functions, such as network creation, simulation and analysis are implemented as plug-ins, which makes it easy to customise and extend the functionality, and it has a sophisticated 3D graphical interface for the creation and editing of neuron and connection groups and for control over simulation and archiving. The simulator is written in C++ using Qt for the GUI.

2.4 Related Work

Here we briefly mention past work that we feel is most relevant to the development of iSpike.

In [13] Gamez et al. describe the design of the CRONOS humanoid robot and the SIMNOS simulation tool for the robot. The sensory joint angle data produced by the robot is converted into a sequence of neural spike patterns using a hybrid of rate based and latency based coding schemes. The produced spike patterns are then delivered to the Spikestream neural network simulator.

[6] by Bouganis et al. describes a spiking neural network architecture that autonomously learns to control a 4 degree-of-freedom robotic arm after an initial time period of motor babbling. Here, joint angles are encoded into neural spike patterns using a variant of population coding and a weighted average of a vector of direction tuned neurons is used to convert a spike pattern into a joint angle for motor control purposes.

Int [30], Novellino et al. describe a bidirectional neural interface used to form a closed loop between in vitro neurons, extracted from rat embryos and plated on a micro-electrode array, and Braitenberg "explorer" two wheeled robot. The interface provides a platform for testing and evaluating different coding and decoding strategies. The supported coding methods are: rate based coding and binary coding, where a spike is produced if the intensity of the analogue signal is greater than a given threshold. A winner takes all mechanism is used to control the robot. While this interface is similar in function to what we are trying to achieve with iSpike, the implementation is hardware based, and hence not easily available or extensible. The library of supported encoding schemes is limited and there is no aspect of biological plausibility.

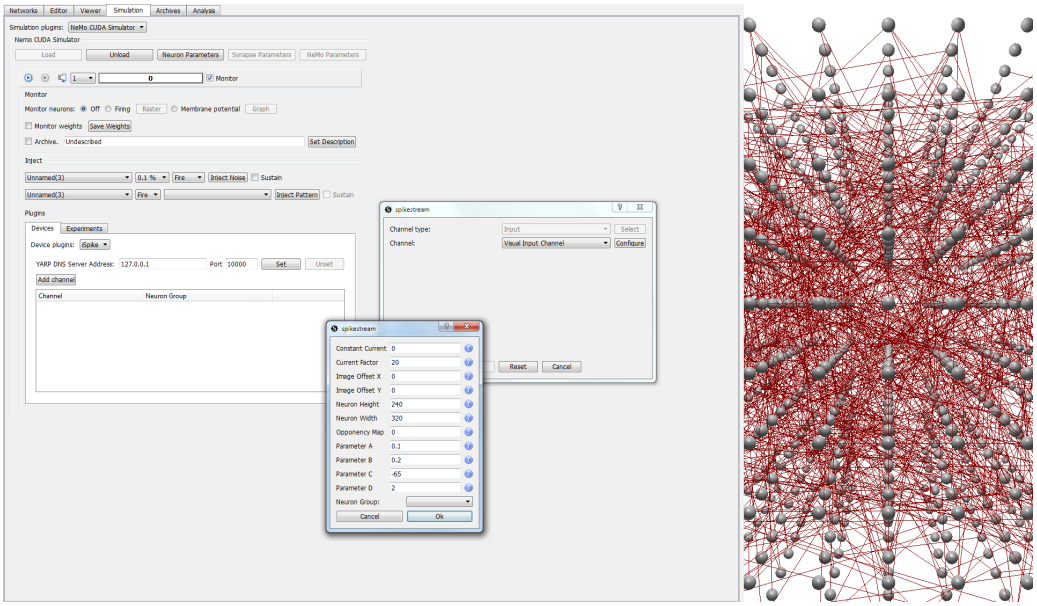


Figure 2.19: SpikeStream GUI.

3 Design

In this section we provide an overview of the design decisions we made initially and the reasoning behind them. We explain the planned high level work-flow of iSpike and the general features we set out to implement. An overview is provided of the process for encoding sensory information into a spike pattern that can be used by a Spiking Neural Network simulator and the reverse process of converting a spike pattern into a sequence of motor commands executable by the iCub robot.

3.1 Implementation Scope

The first design consideration we needed to make early was the scope of the implementation we could realistically aim for. Potentially, iSpike could be a very general purpose tool that could be applicable to any task where conversion between analogue and spike pattern information is needed. For example, instead of using a robot as the source of sensory information, one could easily use any software application, such as a video-game or a stock price monitoring solution.

On the other hand, we had to decide how wide the support will be initially for a variety of Spiking Neural Network simulators. Again, potentially iSpike could be compatible with a variety of simulators that each support different formats for representing a spike pattern and vary in terms of functionality offered.

After looking at the development time that was available, it was decided that the most sensible solution would be to make the core architecture of iSpike as extensible as possible, but only to concentrate on the specific application at hand (using the iCub robot and the NeMo SNN simulator). This would firstly ensure that, by concentrating on a concrete application we would have a working demonstration by the deadline and, due to the extensibility of the architecture, iSpike could later be extended for other use cases if needed.

To ensure the core architecture is extensible and useful for general purpose applications, it would have to satisfy the following constraints: It would have to consist of a *framework of interchangeable components*. Specifically the implementations of the source of the analogue information and the source of a spike pattern should be *abstract* in principle and in our implementation we would implement a single *concrete* implementation of each. The rest of the architecture should make little to no assumptions about the nature or behaviour of the information sources or destinations. iSpike itself should use an *internal representation* of each type of data supported and the concrete implementations should be responsible for converting external to internal data representation.

3.2 Communication with the SNN simulator

An early design decision we had to make was the method we should use to communicate with the SNN simulator. As mentioned earlier, the iCub robot uses YARP for all communication and transfer of information. A YARP nameserver is created at runtime that holds entries of every single iCub input or output port available. Hence all communication with the iCub would happen by using the YARP protocol. For the communication with the SNN simulator we had three options:

3.2.1 Using the existing YARP nameserver

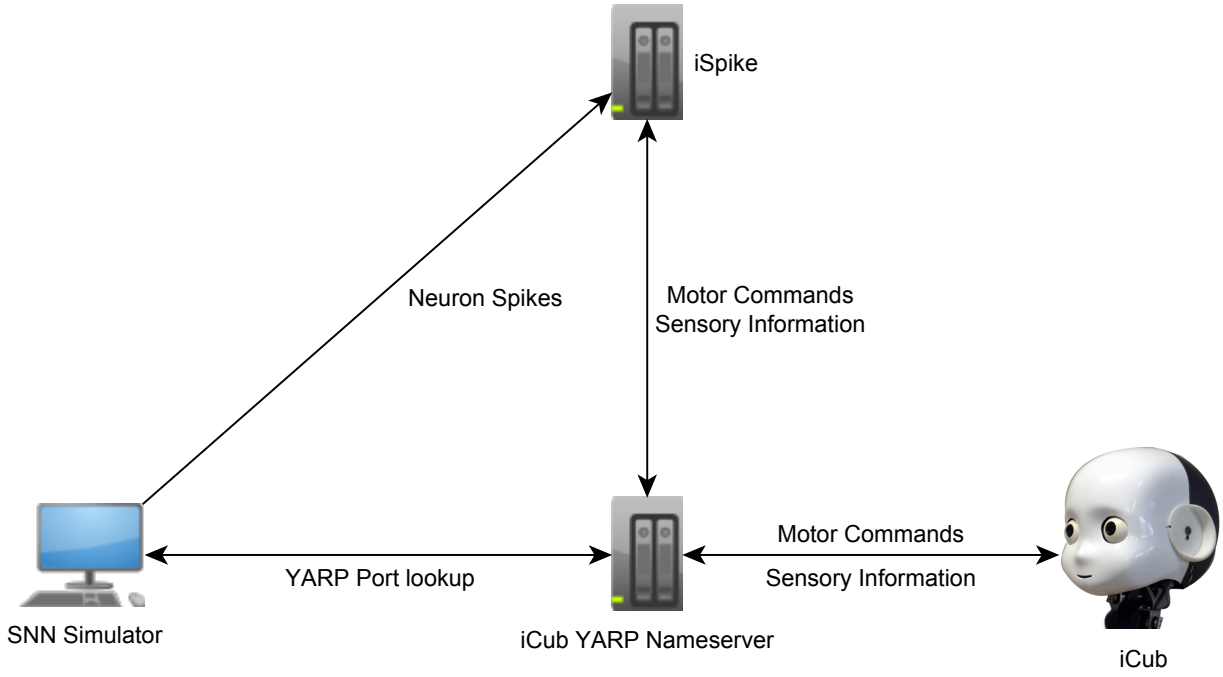


Figure 3.1: Using the existing YARP nameserver

This approach makes the most use of existing infrastructure and involves sharing the YARP nameserver used by the iCub. In this case, iSpike would use the existing YARP ports for communication with the iCub and would register a new set of ports for communication with the SNN simulator. For example, to encode a visual image received from the iCub, iSpike would first find the relevant YARP port using the YARP nameserver, then iSpike would connect to the YARP port responsible for the camera feed and would proceed to retrieve images. These images would internally be converted into a spike form representation and these representations would be made available by a custom YARP port, created by iSpike on the existing nameserver. An SNN simulator would then have to connect to that YARP port in order to retrieve the generated spike patterns. Similarly, for transferring a spike pattern from the SNN simulator to iSpike, a different YARP port would be used on the same nameserver. This approach is shown visually in Figure 3.1.

An obvious advantage to this approach would be the fact that we are using existing infrastructure

which we can assume exists already. This approach is less complex in implementation as the functionality of iSpike is limited to information conversion and communication with YARP. A big disadvantage to this level of YARP integration is that we assume the SNN simulator is itself integrated with YARP and capable of communicating with YARP ports, which none of the currently available SNN simulators are. Also, by using the existing nameserver we might, unintentionally impact the behaviour of other processes using the same nameserver or be, in some way, impacted ourselves.

3.2.2 Using a custom YARP nameserver

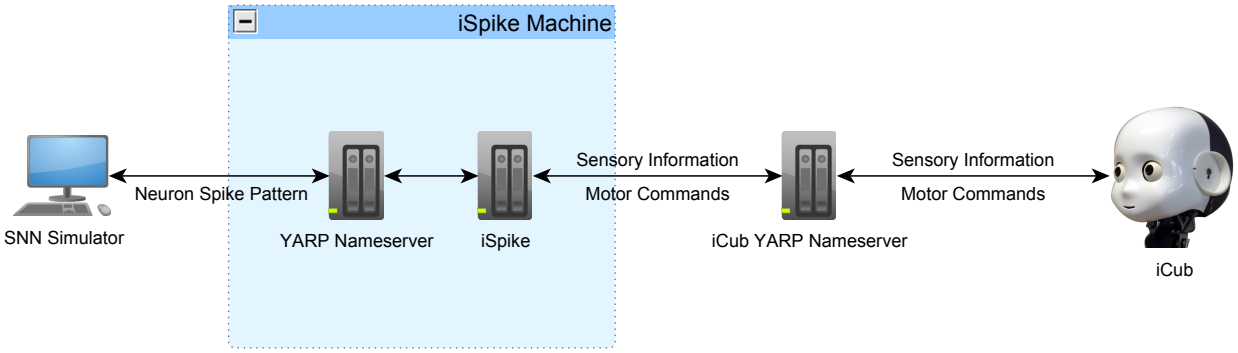


Figure 3.2: Using a custom YARP nameserver

This approach is fairly similar to the previous one. In this case iSpike would create a dedicated YARP nameserver for all it's communication with the SNN simulator. iSpike would receive analogue information and deliver motor commands with the use of iCub's nameserver and would receive and deliver spike patterns to the SNN simulator with the use of a dedicated iSpike nameserver. This approach is visually shown in Figure 3.2

By having a dedicated nameserver we isolate ourselves from any risk that might arise due to sharing a nameserver with the iCub. We are likely to have little to no impact on other processes working with the iCub in parallel to iSpike. We also don't need to worry about issues like port name conflicts on the nameserver. The SNN simulator would still need to be integrated with YARP to realise communication with iSpike. Additionally, this approach adds a level of complexity by requiring iSpike to create and maintain a fully functional YARP nameserver, which would take up some amount of the available system resources which would be better used in the conversion process.

3.2.3 The Library Approach

A third way to communicate with the simulator would be to deploy iSpike as a shared library and let the SNN simulator access the API directly. iSpike would export functions that enable retrieval and delivery of spike patterns and the SNN simulator would call these as and when appropriate. iSpike would still communicate with the iCub's nameserver for all sensory and motor command information. In this case an internal list of ports or communication channels would have to be maintained by iSpike to enable concurrent processing of multiple information feeds. A visual portrayal of this approach is shown in Figure 3.3.

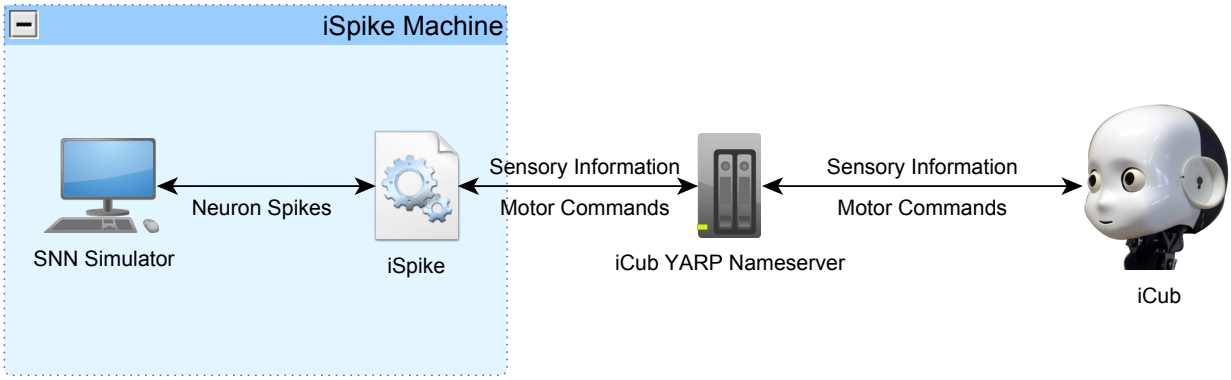


Figure 3.3: iSpike as a library

An advantage to this method is that we gain extra efficiency by communicating with the simulator directly. When using YARP there is extra overhead when information is transferred over the network. The simulator would no longer need to integrate with YARP, in fact the simulator need not know anything about YARP at all. All communication would occur via the exported API. One disadvantage of this approach would be that the source code of any SNN simulator iSpike would support would need to be modified to make use of the published API. Also iSpike would have to reside on the same PC as the SNN simulator, which could potentially be a problem as both are likely to consume significant system resources. Finally, as mentioned earlier, an internal system of ports or communication channels would need to be maintained, which makes development and maintenance more complex.

Our final decision was in favour of the library approach, as we felt that by doing so, we would come closer to meeting the extensibility requirements. By having a shared library and a clearly defined API, documenting how the library should be used, allows the developers of an SNN simulator to decide the level of integration they would like with iSpike and the features they plan to use. A disadvantage of iSpike taking the form of a shared library is that it would limit our potential user-base to developers capable of integrating iSpike with a given simulator. We nonetheless felt that this was the route to take, as any alternatives would be either be less adaptable to other use cases or would be far more complex in implementation.

3.3 Efficiency Considerations

We also looked at iSpike in terms of efficiency we were looking to achieve. We felt there were to main decisions to be made here, which we saw in terms of *real-time* versus *non-real-time* and *synchronous* versus *asynchronous* execution.

Real-time execution implies that the rate of conversion executed by iSpike should at least match the rate at which analogous and spike based information can be retrieved and transferred. For example, if the iCub can provide visual information at 30 frames per second, iSpike should be capable of querying the feed, converting it into spike form and delivering to NeMo at least 30 times a second. If the rate is lower, some frames made available by the camera feed are lost and cannot be used by the SNN simulator. On the other hand, if the SNN simulator can provide us spike patterns at a given rate, we

should be able to retrieve them and convert into motor commands at a similar rate, otherwise some of the patterns are lost and those commands would not be executed by the iCub.

By *synchronous* execution we mean that, at any given point, the conversion process is synchronised with the simulator and the robot. For example, if iSpike is synchronised with the iCub, no conversion will occur, until triggered by the arrival of new sensor data from the robot. Once this happens conversion occurs, then we wait for further data. If a motor control is added to the loop, iSpike will only issue a new command to the iCub when new information is received from the sensors. If iSpike is synchronised with the simulator, new spikes are only delivered and retrieved when triggered to do so by the simulator.

By contrast, in the case of asynchronous execution, iSpike's conversion loop is only controlled by iSpike itself, and does not depend on external triggers. Essentially, we retrieve, convert and deliver information as quickly as possible. If the rate of conversion is higher than that at which new information is made available, we would have to divide the existing information to continuously process new data, or we would just continuously deliver previously processed information until new data is made available.

The main advantage of having synchronous execution is that it is much simpler in design and execution, there are fewer edge case scenarios that need to be considered (for example higher conversion rates than the rates at which new information is made available) and the implementation itself is less complex and more streamlined. On the other hand, asynchronous execution allows much higher conversion rates and would bring us much closer to real-time execution.

Ultimately, it was decided that, given the amount of development time available, it was safer to first implement asynchronous execution and then, if time permits, attempt to implement the conversion in an asynchronous manner. In terms of real-time execution, we decided that we should mainly concentrate on the rate at which the iCub can deliver sensory data and receive motor commands. We would not pay much attention to the simulator's rate of execution, as it is not fixed (as is the case for the iCub), but instead, varies greatly with the size of the neural network.

3.4 Portability Considerations

It was decided that it would be important for iSpike to be a cross platform solution to increase the potential target audience. This especially made sense, as the iCub software, YARP and NeMo are all cross platform and can be run on Windows, Linux and Mac OSX Operating Systems. For this consideration to be compatible with our earlier decision of releasing iSpike as a shared library, we would have to make use of a build system that allows cross platform generation of executables and libraries. A number of such systems exists, a comparison of these and the actual choice we made is available in the Implementation section of the report.

3.5 Biological Plausibility Considerations

One of our initial aims with iSpike was to have as much biological plausibility as we could implement, given the time constraints. For this, we had to look at the available models that exists for simulating

the biological information conversion pathways, which we described briefly in the Background section of the report.

3.5.1 Feed Quantity

Before deciding on the level of biological plausibility to strive for, we had to deduce the number and types of supported data formats by iSpike. iCub, after all, provides a large variety of sensory stimulus, including visual feeds, audio feeds, joint angle information, information relating to balance. We could not realistically aim to support all of these, so a decision had to be made on the number of sensory feeds to support in the initial build of iSpike.

It was assumed, that the support for visual feeds would be important, as this allows rich inputs to the neural network in regards to the surroundings of the robot and would allow the end user to use iSpike for solving a variety of interesting problems, including visually aided locomotion, object recognition and saliency. The visual pathways, while complex in nature, have also been well researched and a variety of models exist (as mentioned earlier) to simulate the behaviour of the individual components that this pathway consists of.

Another sensor feed, that we felt had to be included in the initial build, was that of Joint Angles. Without the support for receiving and transferring information about individual joints, we could not create a system capable of executing any level of control over the robot, so at least some support for joint information conversion and motor command delivery had to be implemented.

We decided to limit ourselves to only these two types of information provided by the iCub: *visual feeds* and *joint angles* and strive for higher biological plausibility in the conversion process of these feeds, as opposed to the support of a multitude of feeds, where each would be poor from either implementation or biological plausibility perspective.

3.5.2 Model Choices

Having decided on the sensor feeds we will support, the next step was to decide the initial models to use for implementation. For efficiency reasons and to reduce complexity of implementation, we only simulate the behaviour of photoreceptors and the ganglion colour opponent cells. We ignore the behaviour of Bipolar cells, as their contribution to the transformed image is not clear. Apart from the colour opponent Ganglion cells, there are others, for example cells that respond to light intensity. We exclude these to simplify the initial implementation, but plan to add them once the initial implementation is successful.

Foveation

As soon as we receive a regular, 2-dimensional image from the iCub, we need to foveate it to reflect the distribution of photoreceptors in the human retina. As discussed earlier, the choice we have at this point is to use the *log(z)*, the *log(z+a)* or the *Wilson's Model*. Their biological plausibility follows the same order we introduce them, with the log(z) model being least biologically plausible due to the uniform distribution of photoreceptors near the center and non-overlapping rectangular receptive fields. Wilson's Model is the most biological plausible. In terms of complexity and ease of implementation, the order is inverse: The log(z) model is easiest to implement, as all that is needed is

to sample the image using log-polar coordinate system. For the Wilson's model, we need to consider each concentric receptive field in turn to evaluate the pixels it contributes to. This is more complex to implement and is slower in execution. By using Wilson's model it would take longer to create a foveated view of the original image, which would make it more difficult to achieve our aim of real-time execution. For these reasons, we decided to use the simpler log(z) model for foveation due to it being less complex and easier to implement whilst still being, to some degree, biologically plausible.

Ganglion Colour Opponency

For simulating the Ganglion colour opponent cells, we decided to use the algorithm explained in the relevant background section. The principle is to use a difference-of-gaussians filter on the image with varying intensities to emulate the behaviour of the ganglion cells as shown by Figure 2.11. This model was originally referenced in the 1966 research paper by Enroth–Cugell et al. [9] and at this point no other viable alternative models are available. Difference of Gaussians filters are also computationally a good choice, as there exist efficient algorithms to perform the transformation, such using sequential 1–dimensional Gaussian blurs instead of a single 2–dimensional blur.

3.6 Summary of Design Decisions

Here we would like to summarise the considerations and decisions made in the design phase of iSpike. By considering the potential scope of iSpike and the given development time, it was decided that we should concentrate on using iSpike with the iCub robot and the NeMo SNN simulator *only* while trying to make the core architecture extensible enough to add support for other robots/simulators. To be accessible and extensible, iSpike would have to be *component based* and adding a new component should have little impact on the rest of iSpike. iSpike itself, should be a shared library, used by the SNN simulator due to the potential gains in efficiency. The initial implementation should be *synchronous* as this is a more realistic target, given the time available. iSpike should strive for real-time performance in the sense that the rate of data conversion should at least meet the rate at which information is made available and processed by the iCub.

In terms of biological plausibility it was decided that in the initial build only *visual* and *joint* information should be supported, mainly due to time constraints. For the visual feed, the *log(z)* model should be used with the *Difference-of-Gaussians* as an approximation for the colour opponent cells functionality.

3.7 Conversion Process

At the core of iSpike is the conversion process between analogue sensory data and neuron spike patterns and the reverse conversion from a neuron spike pattern into an executable motor command. Here we provide a high level view of the conversion processes to be supported in the initial version of iSpike.

3.7.1 Visual images to spike patterns

The task of this process is to convert an ordinary 2–dimensional image received from one of the iCub's cameras into a pattern of neuron spikes that can then be used by an SNN simulator. As mentioned earlier, the first transformation process would be to apply a log-polar foveation transformation to mimic the distribution of photoreceptors in the retina. The output would be a log-polar representation of the original image with higher resolution near the central part and increasingly lower resolution near the periphery.

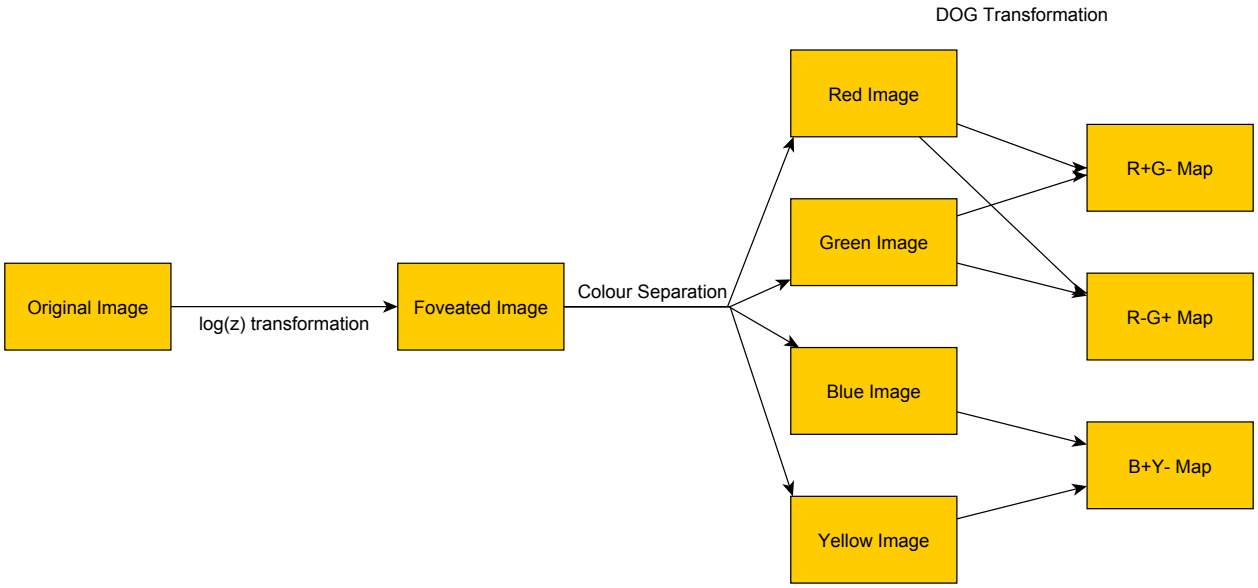


Figure 3.4: First part of visual feed conversion into a spike pattern

Afterwards we would separate the log-polar representation into the individual colour components (combine Red and Green images to get Yellow) and apply a Difference Of Gaussians transformation to pairs of these images to receive a monochrome colour opponency map, as shown in Figure 3.4. Figure 3.5 shows a visual representation of the image preprocessing stage.

The next step would be to create a 2–dimensional array of Izhikevich neurons with the same dimensionality as the colour opponency map. We need a matrix of Izhikevich neurons to produce a spike map in the end. Each neuron maps to a single pixel in the opponency map. We then feed a current to each neuron that is proportional to the intensity of the pixel the neuron is responsible for. As a result of feeding current to the neurons, some neurons will spike if the current is high enough. We can then observer which neurons have fired in response to the given image and use that as our spike pattern, usable by the SNN simulator. This part of conversion is shown in Figure 3.6.

3.7.2 Joint angles to spike patterns

The next conversion process we look at is attempting to convert an observed joint angle into a spike pattern usable by the SNN simulator. To do this, we plan to utilise a 1–dimensional array of Izhikevich spiking neurons and use *population coding* to encode a single angle. Using a population code here is

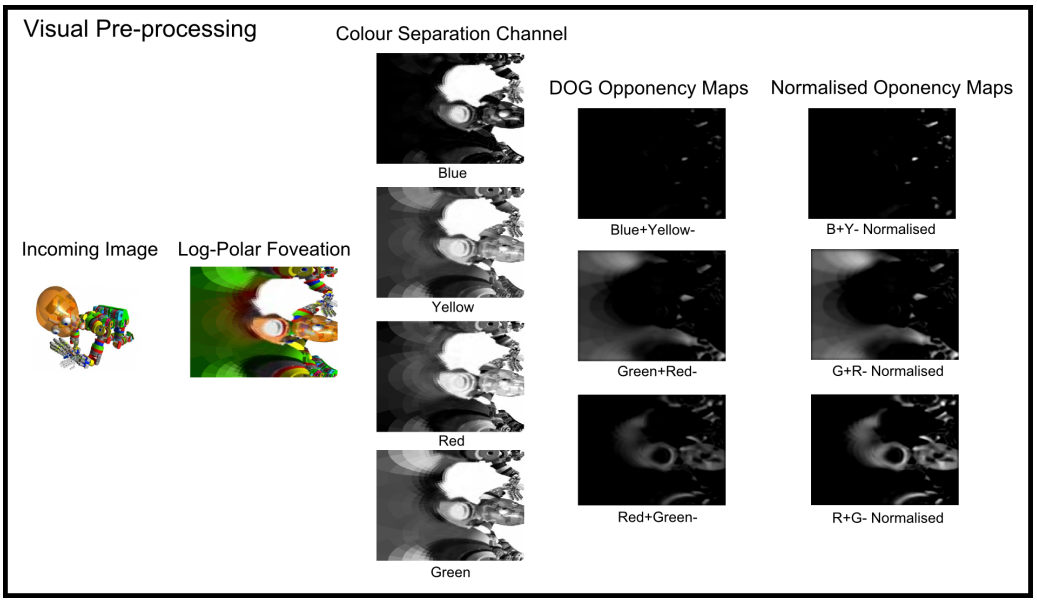


Figure 3.5: Visual representation of image preprocessing

the most direct approach of increasing the dimensionality of a single observed angle. An alternative would be to use a single neuron coupled with a *rate* based or *latency* based code, but this is likely to show high fluctuations in value and a high error rate due to low dimensionality.

We begin by creating a 1–dimensional structure of neurons. The dimensionality could be chosen by the end user. In order to produce a spike map, we need to feed current to the neurons. The individual current values for each neuron are produced by applying a Gaussian function with mean equal to the observed angle and standard deviation, again, provided by the end user. The neurons we created earlier are assigned angles in a given angle range, so that, each neuron is *sensitive* or *responsible* for a single angular value. We then evaluate the Gaussian function at each of the angular values to receive a current value. We upscale this value, as otherwise it would be in the range between 0 and 1, which is not sufficient to cause any firing in the neurons. These current values are then fed to the neurons created earlier, allowing us to observe a spike pattern as neurons, that receive enough current, start to fire. This spike pattern can then be delivered to the SNN simulator. This process is shown in Figure 3.7.

3.7.3 Spike patterns to motor commands

This process is concerned with converting an observed pattern of neuron spikes back into a joint angle, and finally into a motor command for the robot to perform. For this, we use an *inverse of a population code*, we assume the spike pattern was received from a one dimensional array of neurons. The spikes are then downscaled to a single joint angle value. To do this, we create a one dimensional array of receptors with equal dimensions to the number of neurons producing the spike pattern. Each receptor has an associated intensity value and is again responsible for a single angle, in total spanning a range of angles similarly to the previous process. The intensity values increase with each observed spike and decay over time. Thus, with continued input from the spike pattern, the receptors receiving spikes

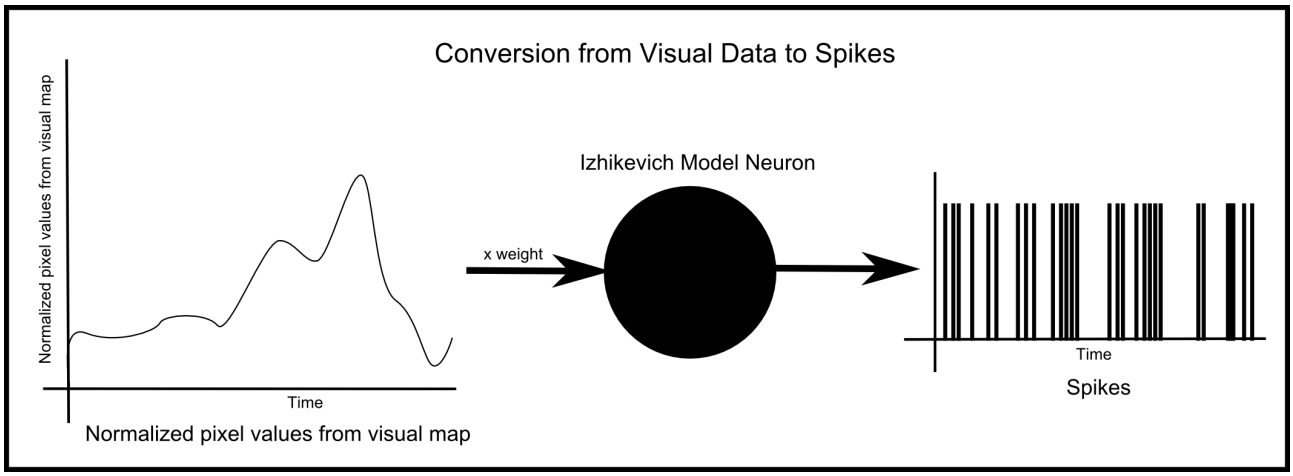


Figure 3.6: Second part of visual feed conversion into a spike pattern

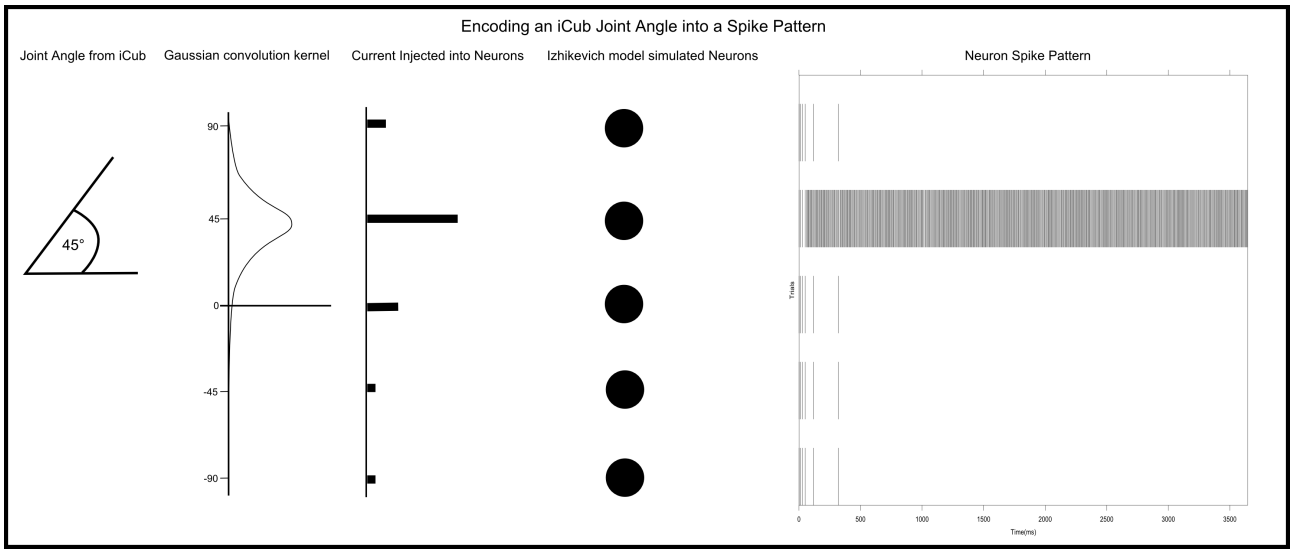


Figure 3.7: Joint angle feed conversion into a spike pattern

more often will have a higher intensity and receptors receiving no spikes will eventual decay to zero.

To retrieve a single joint angle value we take the *weighted average* of the receptor angles with weights being the intensity values. With continuous input from the spike pattern, the weighted average will converge toward a single value. A separate component then takes this single joint angle and inserts it into a motor command, ready for delivery to the iCub. This process is shown in Figure 3.8.

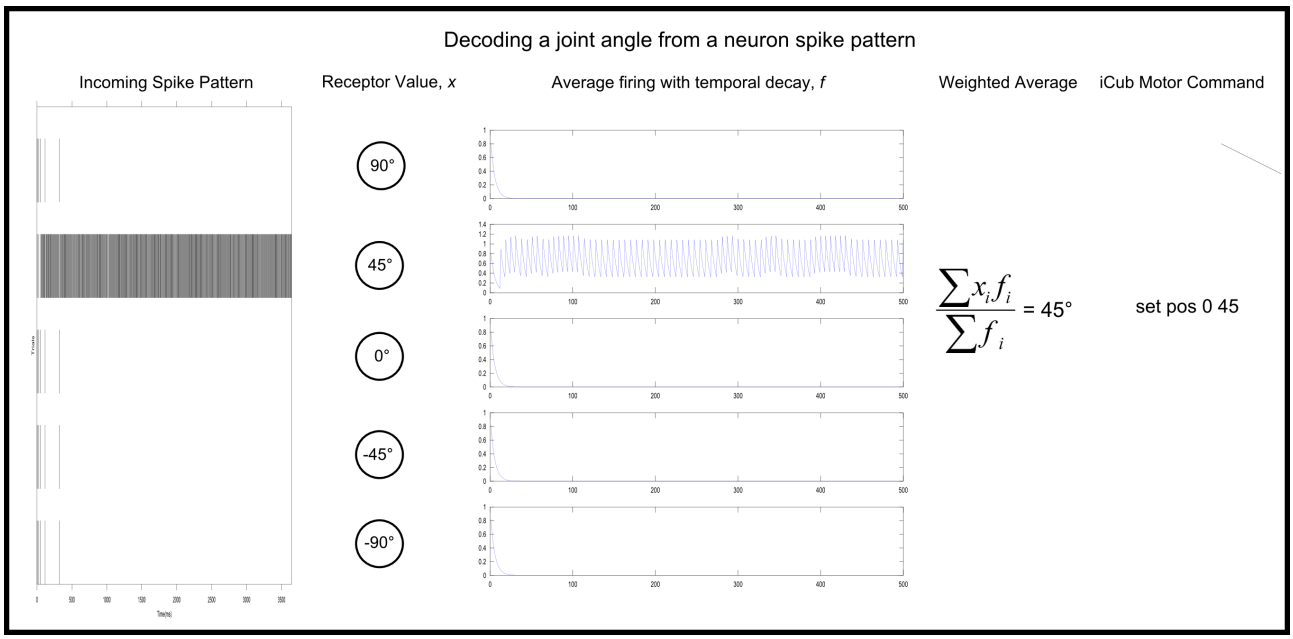


Figure 3.8: Spike pattern conversion into a motor command

4 Implementation

In this section we describe the implementation details of the major components of iSpike. We describe the role each individual component plays in the overall system and give a complete high level overview of the whole architecture near the end of the section.

4.1 Technical Details

We begin by mentioning the software solutions we used and relied on during the development of iSpike. iSpike was developed in C++. The main reasons for choosing C++ were for efficiency and integration purposes, we felt that we might be able to exploit the closer to the system nature of C++ in our conversion process and YARP, the iCub software, NeMo and Spikestream are all programmed in C++, so integration would be easier.

To achieve portability across platforms, we used a build system called *CMake* (Cross Platform Make) and we also heavily depended on the functionality provided by the Boost C++ libraries. We tried to keep the number of dependencies iSpike has as low as possible, to make distribution and installation easier for the end user, so Boost is the only set of libraries iSpike depends on at the moment.

4.1.1 CMake

As the full name suggests, CMake is a cross platform build system that is open source and is compiler independent. CMake can create both binary executables and libraries on a variety of operating systems, including Windows, Linux and Mac OSX. Many development chains are supported, including make (or mingw-make under Windows), Microsoft Visual Studio, Apple's Xcode and others. CMake also provides a GUI application for configuration and build file generation purposes. Some notable projects that use CMake include the Blender and OGRE open source rendering engines as well as the MySQL relational database management system.

4.1.2 Boost Libraries

Boost is a set of libraries created to extend the functionality of C++. Boost release contains over 80 individual libraries, including libraries for linear algebra, multi-threading, logging, unit testing and others. We made the decision to utilise the Boost libraries, as they provide a large variety of functionality and we felt that if we did use Boost, we would not need to depend on any other libraries, which would make it much easier for a developer to build iSpike from source code. Also the Boost libraries are famous for being well written, many of Boost's founders are on the C++ standards committee and some functionality from the Boost libraries is included in the next planned standard for the C++ programming language (C++0x).

4.2 Communication with YARP

YARP provides a shared library with API that enables communication and data transfer across YARP ports. We decided not to use the library provided and instead use access methods detailed in the "YARP without YARP" tutorial available at [11]. We made this decision as we wanted to keep the number of iSpike dependencies to a minimum and felt that we do not need the majority of the functionality the library provides. Moreover, the library uses it's own representation for the data being transferred, for example images, and we decided that the conversion from the representations used by the library to our own representations would cause an unnecessary overhead in development time and during execution.

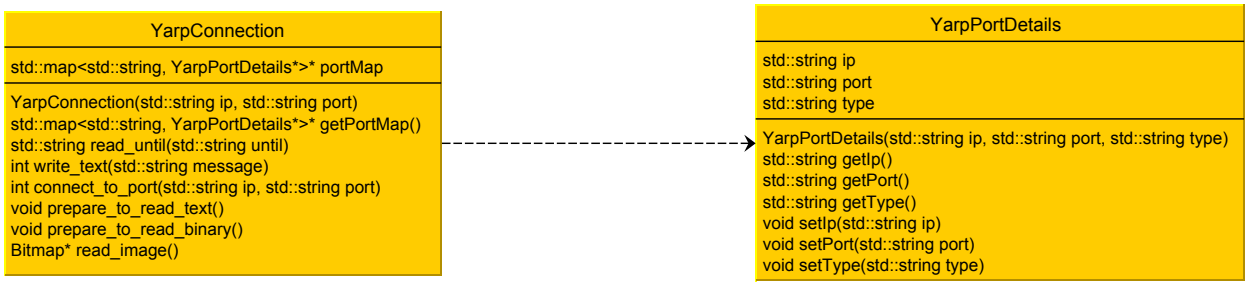


Figure 4.1: `YarpConnection` and `YarpPortDetails` classes

Instead, we followed the tutorial mentioned earlier and created a `YarpConnection` class that exports a list of functions covering all interaction with YARP ports needed for iSpike. We will now describe how we implemented communication with YARP without relying on the YARP shared library. A visual representation of the `YarpConnection` class together with the DTO object we use to store YARP port details `YarpPortDetails` is available at Figure 4.1.

Connecting to a YARP port

The first point of interaction with YARP would normally be the connection to the YARP nameserver in order to retrieve information regarding the available YARP ports. It is assumed the IP address and port of the nameserver are known beforehand. A connection is established by connecting to the appropriate IP address and port and sending the following command:

```
CONNECT iSpike
```

where `iSpike` is a client identifier and can take any value. The server normally replies with the following:

```
Welcome iSpike
```

This means that the connection has been established and we can proceed to send commands.

Receiving Port Details from a Nameserver

After connecting to a YARP Nameserver we can proceed to query it about the available YARP ports, we do so by issuing the following commands:

```
d
list
```

The `d` command signals the beginning of input, the `list` command requests a list of available YARP ports. The reply from the nameserver is similar to the following:

```
registration name /iCub/cam/left ip 10.0.0.1 port 59 type tcp
registration name /iCub/cam/right ip 10.0.0.1 port 60 type tcp
registration name /iCub/head/rpc ip 10.0.0.1 port 61 type tcp
registration name /root ip 10.0.0.1 port 55 type tcp
*** end of message
```

This output contains the name of each YARP port registered on the system with the associated IP address and system port, as well as the connection type. Only a small fragment of the ports provided by the iCub is shown above. We then parse this output and create a `YarpPortDetails` object for every entry and store it in the `portMap` field of the `YarpConnection` object.

Receiving text

Receiving text based data (such as joint angles) is relatively straightforward, we connect to the appropriate YARP port as described earlier and then send an `r` command. This reverses the connection direction and from now on any information sent to the port by the other party is available to us. We only need to continuously read new data as it arrives and parse it.

Receiving images

In order to receive images from a YARP visual port, we first need to connect to the port in binary mode. The connection procedure outlined earlier can only be used to transfer character based information. We connect to the port we are interested in and send the protocol identifier:

```
'Y','A',0x64, 0x1E, 0, 0,'R','P'
```

This lets the server know we would like to use the TCP protocol and would not like the server to send us acknowledgements (the "fast" TCP connection). Next, the server expects us to send *our* port name length and value. As we do not have a port name, we can send any invalid string, as long as it does not start with the `/` character:

```
4,0,0,0,'m','i','n',0
```

Here first four bytes represent the little–endian length of the port name (4) and the next four bytes represent the null terminated name of our port. The server then replies with a confirmation:

```
'Y' 'A' B1 B2 0 0 'R' 'P'
```

Where B1 and B2 identify a socket port number (unused). We can use this reply to verify if the handshake has succeeded. At this point we have connected to the port, provided the connection type to use and have identified ourselves, now the server is awaiting for our commands. We send the following to the server:

```
'Y','A',10, 0, 0, 0,'R','P'
1, 1, 255, 255, 255, 255, 255, 255, 255, 255
8, 0, 0, 0
0, 0, 0, 0
```

The first message is used to let the server know the length of the "index header" (the next message), in our case it is 10. The second message let's the server know how many "blocks" will be used to send the command. This value is encoded by the first byte in the message, in our case it is 1. The third message tells the server the length of our command, in our case the command will be 8 bytes long. The final message provides the length of reply messages, we would not like any reply messages, hence the length is 0. At this point we have provided the length of our command to the server and all that is left is sending the actual command. Each command is of the following format:

```
0, 0, 0, 0, '~', CHAR, 0, 1
```

where `CHAR` identifies the actual command. In our case we would like to reverse the connection in order to receive the images, hence we send the following:

```
0, 0, 0, 0, '~', 'r', 0, 1
```

This reverses the connection and the server will now proceed to send us images in bitmap format with a YARP header attached to each.

Now, for every image sent, YARP will provide a header together with the command `d` to identify the transfer of binary data. The header is identical to the one we sent earlier. We are only interested in extracting the number of "blocks" and the length of each "block" containing the binary information (the actual image). Having extracted the size of the image, we can proceed to read the image contents from the socket and store the results in a buffer. The image itself is a simple 32 bit bitmap of interleaved RGB colour values, 4 bytes per colour.

Sending motor commands

To send a motor command to a given joint, we connect to the relevant YARP port in text mode and proceed with the connection handshake. We send a `d` command to identify the beginning of input. Motor commands are of the form:

```
set pos degreeOfFreedom angle
```

where `degreeOfFreedom` identifies the degree of freedom for this particular joint that we would like to control. For example the single head joint can move up, down, left and right. Each direction is a degree of freedom. `angle` identifies the final angle we would like that degree of freedom to take. For example we might next transfer:

```
set pos 0 20
```

If connected to the YARP port representing the head joint of the robot, this would move the head upward to a 20 degree angle. The movement is performed with the highest possible velocity for that joint.

4.3 Neuron Simulation

As mentioned earlier in the Design section of the report, we utilise Izhikevich neurons internally in the conversion process both for Visual and Joint Angle data. For this we implemented a class called `IzhikevichNeuronSim` which is responsible for creating a one or two dimensional structure of disconnected neurons and providing the identifiers of neurons that have produced spikes in response to an incoming current. Figure 4.2 is a visual demonstration of the class contents.

Variables `a`,`b`,`c` and `d` are Izhikevich model parameters. Variables `v` and `u` are used internally to represent the potential and the recovery variable value for each neuron. The `spikes` structure is used to hold the any neuron spikes that have occurred. The `constantCurrent` variable enables the delivery of constant background current in addition to the one passed normally and the `currentFactor` variable enables scaling the incoming current.

The class constructor initialises the internal variables and creates a network of neurons with the given dimensions. The `getSpikes` function takes a current map as a parameter, delivers this current to the constructed neurons and simulates the network for 1 ms. After this, any spikes that have occurred are returned.

4.4 Readers

In iSpike, we used the concept of a *Reader* to represent an entity capable of receiving sensory data of a *given type* from a *given location*. So, for example, a `YarpAngleReader` would be a class capable of receiving Joint Angle data from YARP.

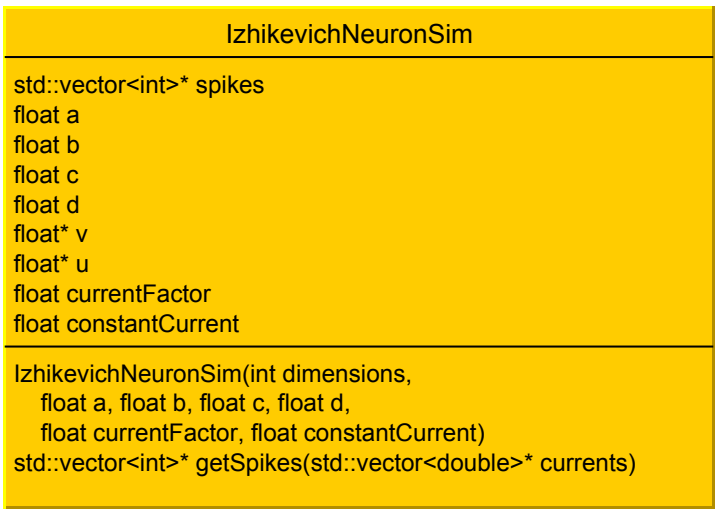


Figure 4.2: `IzhikevichNeuronSim` class

4.4.1 Creating a Reader

All Readers are created via the `ReaderFactory` class (Figure 4.3). This class was created with the Abstract Factory Design Pattern in mind. Having this class, allows the creation of new Readers at runtime with a user selected type and parameters. The `ReaderFactory` constructor method can be called with or without an IP address and port parameter. If the parameters are provided, both Readers that require YARP and Readers that don't can be created using the factory. Without the parameters only Readers that do not require a connection with YARP can be created.

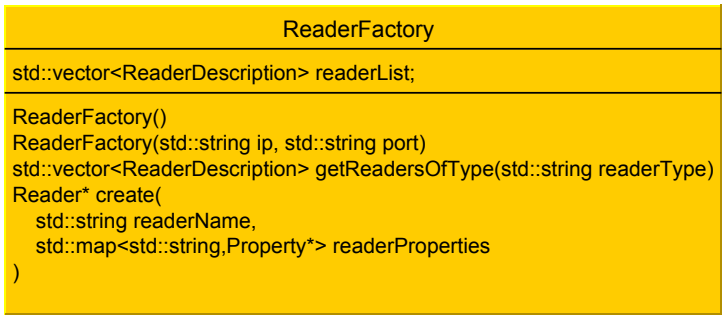


Figure 4.3: The `ReaderFactory` class

The `ReaderFactory` class provides a method to query the list of Readers, of a particular type, that can be constructed. This method enables displaying a list of Readers that can be constructed, and the end user can select an item from the list to construct that Reader. Readers are divided in types according to the type of information they gather, for example, "Visual Reader" or "Angle Reader".

Having chosen a desired Reader, the `create` method of the `ReaderFactory` class can be called to create the desired Reader. This method accepts two parameters, first is the name of the Reader

to be created, second parameter is a filled in map of parameters that the new Reader is initialised with. The list of supported parameters is available from the `ReaderDescription` structure (Figure 4.4) returned by the `getReadersOfType` method. The `create` method returns a pointer to the newly created Reader.

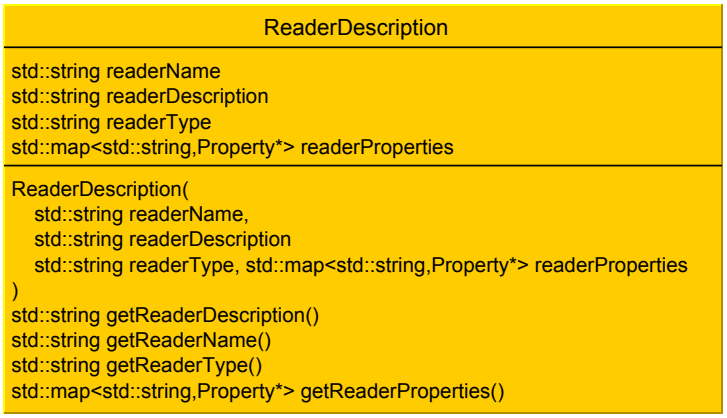


Figure 4.4: The `ReaderDescription` class

4.4.2 Reader at runtime

Once a Reader has been created, it can be started via a `start` method. This method is supported by all Readers and essentially creates a new internal worker thread where the information retrieval loop resides. As the loop executes, new information is retrieved and put in a `Vector` of data structures, depending on the type of the Reader. Each Reader has a `getData` method, that retrieves the vector of data structures that have been accumulated since the last call to this method. In the case of a `VisualReader` the data structures would be images, in the case of an `AngleReader` they are joint angles. Figure 4.5 shows the work-flow involved in creating and executing a Reader.

4.5 Supported Readers

4.5.1 File Angle Reader

The `FileAngleReader` is the simplest of the currently implemented Readers. As the name suggests, this class uses a file as the source for Joint Angles. This class has a single parameter, which is the filename of the file to read the angles from. Once started, the worker thread will attempt to open the file and will store the read angles in a local buffer. They can then be retrieved by calling the `getData` method. The file contents are expected to be of the form:

```
0 0 50 0
```

where each entry represents the angle of that degree of freedom. Here, 4 degrees of freedom are available and the angle of the 3rd is 50 degrees. This Reader is mainly used for debugging purposes.

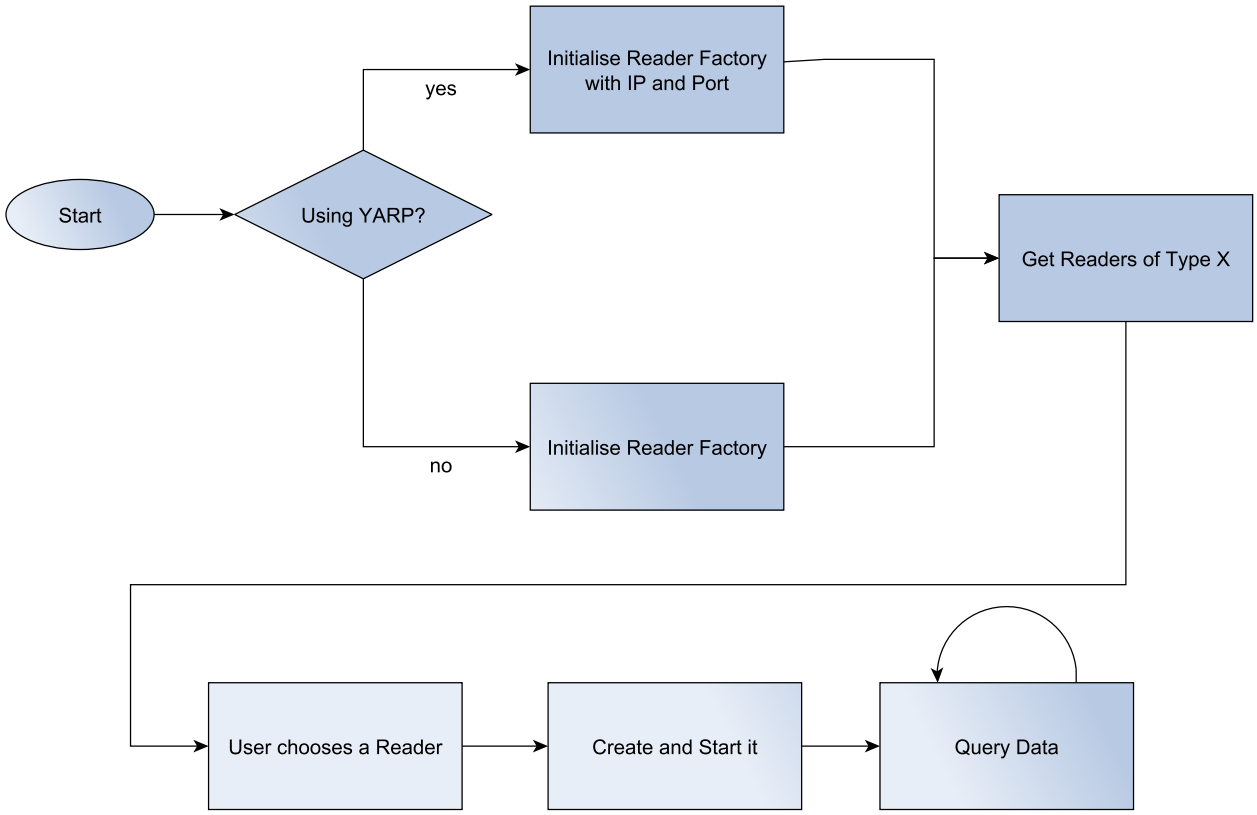


Figure 4.5: Creating and running a Reader

4.5.2 File Visual Reader

The `FileVisualReader` class is a type of Visual Reader that takes it's contents from an image file on the local hard disk. This Reader also has a single parameter, which is the file name of the image file to be used. The image file is expected to be of PPM (Per-Pixel Map) format, which is one of the most basic image formats available. The contents of the file are interleaved RGB values of the individual pixels. The image header consists of the format identifier (P1 - P6) and the width and height of the image. Many graphics applications support the PPM format, for example GIMP and IrfanView.

4.5.3 YARP Angle Reader

The `YarpAngleReader` class is capable of retrieving a list of joint angles from a particular joint via a YARP port. The only parameter of this reader is the name of the YARP port for the joint we are interested in. Upon initialisation, this Reader will connect to the YARP nameserver (the IP address and Port are provided by the `ReaderFactory`) and find out the IP and Port of the joint parameter. During execution, the `YarpAngleReader` will attempt to connect to the IP and Port retrieved during the initialisation phase and will then proceed to read the available joint angles in a loop. The Reader will retrieve a joint angle for every Degree of Freedom for that joint and will store these in an array buffer. The user can the retrieve the collected angles by calling the `getData` method.

4.5.4 YARP Visual Reader

The `YarpVisualReader` retrieves image information from a YARP port at runtime. Again, the only parameter of this Reader is the YARP port name to read the images from. The rest of the information regarding the image (width, height, depth) is provided by the YARP protocol at runtime. During initialisation this Reader will, similarly, query the YARP nameserver to find out which IP address and port can be used to connect to the image feed. At runtime, this Reader will connect to the IP address and port retrieved earlier and will proceed to read images as they become available on the communication channel and store them in a local buffer, where they can be retrieved by calling the `getData` method on this Reader.

4.6 Writers

We used the concept of a *Writer* to denote an entity capable of outputting data of a *given type* to a *given destination*. For example, a `YarpAngleWriter` would be a process capable of outputting joint angle motor commands to a given YARP port.

4.6.1 Creating a Writer

All Writers are created via the `WriterFactory` class, which is analogous to the `ReaderFactory` described earlier. A visual overview of the class is available in Figure 4.6. There are two constructors available, if the IP address and port parameters are provided, this `WriterFactory` can be used to create YARP enabled Writers, otherwise only local Writers are enabled.

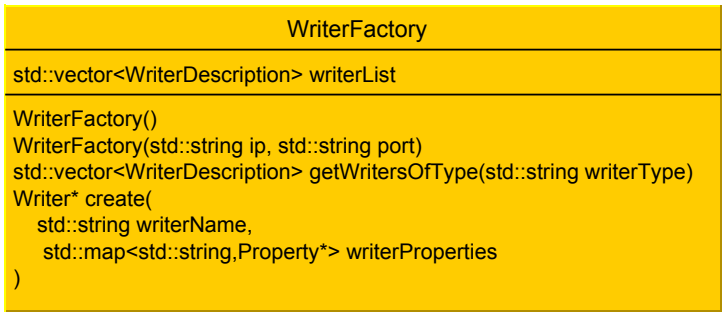


Figure 4.6: The `WriterFactory` class

After constructing the class, a list of available Writers of a particular type can be retrieved via the `getWritersOfType` method, where type identifies the type of data the Writer can output. Currently only the "Angle Writer" type is available, which identifies all Writers capable of outputting motor commands for individual joint control.

Having chosen the desired Writer, the `create` method can be called to instantiate a writer of a given type and with the provided parameters. The parameters for a particular Writer are available via a `WriterDescription` Data Transfer Object associated with each Writer. Normally the user would retrieve the default parameters of the desired Writer, make amendments to this structure and then

instantiate the Writer with the updated parameters. A visual overview of the `WriterDescription` class is available at Figure 4.7.

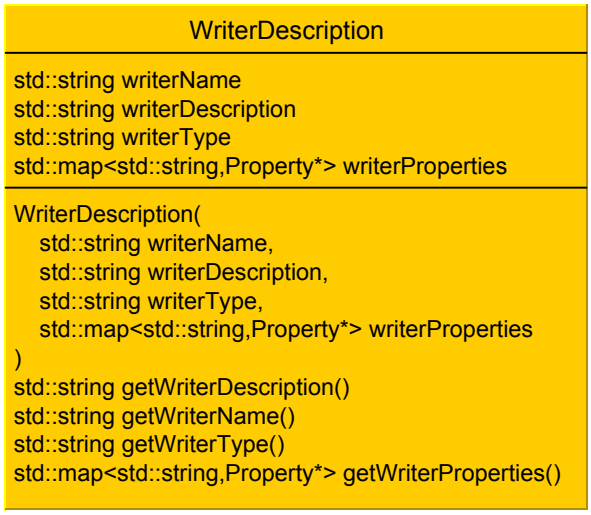


Figure 4.7: The `WriterDescription` class

4.6.2 Writer at runtime

The execution of a Writer is initiated by a call to the `start` method, which every Writer provides. This method starts the worker thread, which goes continuously goes through an internal stack buffer and sends one command from the stack to the output. If the stack becomes empty, the Writer waits for more information. Different types of Writers provide different means of adding a new entry to the internal buffer. Writers extending from the `AngleWriter` abstract class, provide an `addAngle` method, which adds a single angular value to the internal stack. Figure 4.8 illustrates the inner loop of an Angle Writer.

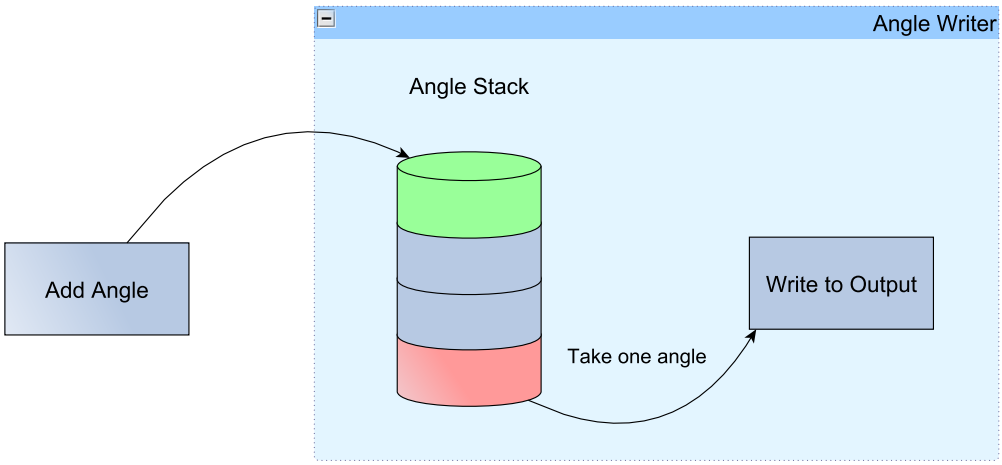


Figure 4.8: Inner loop of an Angle Writer

4.7 Supported Writers

4.7.1 File Angle Writer

The `FileAngleWriter` class is concerned with receiving Joint Angles from the user and outputting them to a file on the local hard disk. The only parameter of this Writer is the filename of the output file. In every loop iteration, provided that the *angle stack* is not empty, the Writer will take one set of angles (one angle per degree of freedom) and append them to the file. If the *angle stack* is empty, the `FileAngleWriter` will wait for further input.

4.7.2 YARP Angle Writer

The `YarpAngleWriter` class iteratively takes an angle from the *angle stack* and sends a motor command to a given YARP port, that causes that joint to perform movement to the given angle. This writer has two parameters. The first parameter is the YARP port name of the joint to be controlled. The second is an integer value representing the degree of freedom for that joint that should be controlled. Upon initialisation, the Writer will connect to the YARP nameserver to resolve the port name. At runtime the Writer will connect to the resolved location and continuously provide motor commands causing a movement in the joint, provided that the *angle stack* is not empty.

4.8 Input Channels

We use the concept of an *Input Channel* to identify a process capable of receiving sensory data from a *Reader of a given type* and transforming that data into a neuron spike representation, which can then be used by an SNN simulator. An Input Channel would normally be associated with a Reader that provides data for the Input Channel to use. For example, a `JointInputChannel` converts the angular data prepared by an `AngleReader` (be it `FileAngleReader` or `YarpAngleReader`) into neuron spikes.

4.8.1 Creating an Input Channel

All Input Channels are created via a class called `InputChannelFactory`. This class is like a catalogue of available Input Channels and can provide a list of all available Input Channels, it can also create and initialise a new Input Channel instance. Figure 4.9 shows the `InputChannelFactory` class along with the `InputChannelDescription` DTO that this class uses.

A user would typically create a new instance of `InputChannelFactory`, then call the `getAllChannels` method. This method returns a vector of all available Input Channels. The user can then select the channel he or she is interested in. After this, a `ReaderFactory` can be used to create a Reader of the appropriate type (which can be known by calling the `getReaderType` method on the `InputChannelDescription` DTO for that channel). With the Reader created, a new Input Channel can be created and initialised by calling the `create` method of the `InputChannelFactory` class. This method will use the provided channel name, the Reader created earlier and the provided properties to create and initialise a new Input Channel of the desired type.

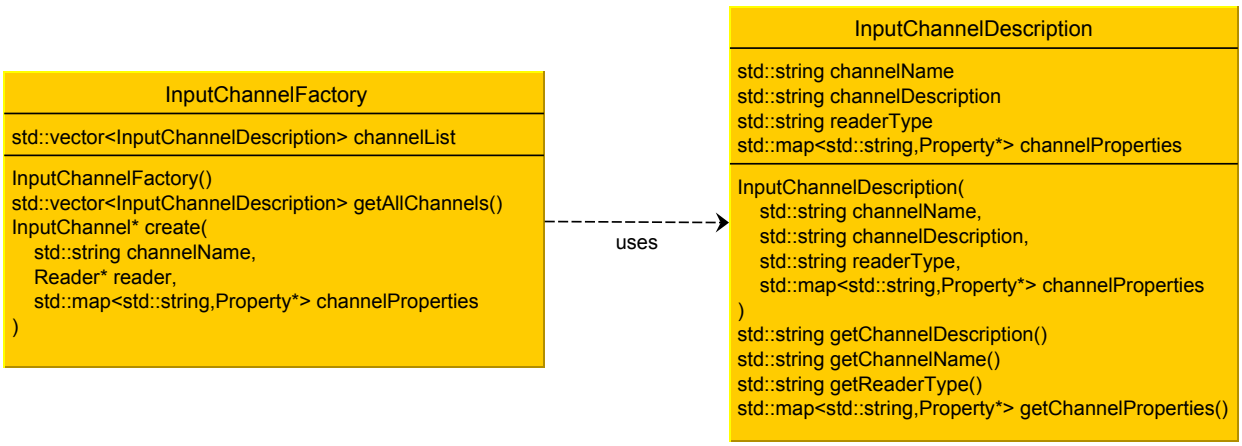


Figure 4.9: The `InputChannelFactory` class and the `InputChannelDescription` DTO

4.8.2 Input Channel at runtime

The newly created Input Channel can then be initiated by calling the `start` method, which is provided by all Input Channels. This method will start the associated Reader, causing it to start gathering data. Then the worker thread of this Input Channel will be started. The worker thread normally consists of the conversion loop, which acquires new information provided by the Reader, converts it into a spike representation and adds it to the local buffer. The buffer is a stack of spike patterns, the contents of which, can be retrieved by calling the `getFiring` method. Since all Channels are synchronised (due to reasons explained in the Design section of the report), the execution loop of the worker thread halts at the end of the loop and waits for the user to call the `step` method, which executes the loop once more. This technique allows controlled execution and enables synchronisation across channels. Figure 4.10 illustrates the behaviour of a typical Input Channel worker thread.

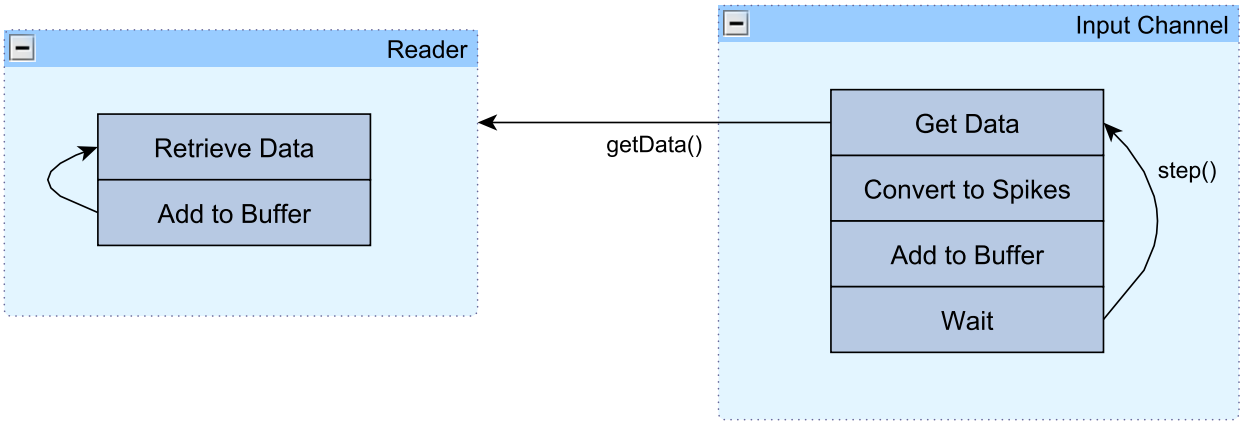


Figure 4.10: The worker thread of a typical Input Channel

4.9 Supported Input Channels

4.9.1 Joint Input Channel

A `JointInputChannel` class is capable of retrieving a Joint Angle from an associated `AngleReader` and converting it into a spike pattern. The method for encoding has been described in the Design section of the report. The following is a pseudo-code rendering of the worker thread loop for a `JointInputChannel`:

```
double standardDeviation = calculateStandardDeviation();
vector_of_double angles = reader.getData();
if(new_angles_received) {
    for each neuron {
        double currentAngle = interpolateAngle();
        currents[neuron] = getPDF();
    }
    vector_of_int spikes = neuronSim.getSpikes(currents);
    buffer.addSpikes(spikes);
}
    waitForStep();
```

As the pseudo code snippet above shows, we initially calculate a standard deviation for the Gaussian curve we fit later on the neurons. According to statistics, for a Gaussian curve, values three standard deviations from the mean in each direction cover 99.7% of the values. We allow the user to provide the standard deviation parameter of this channel as a percentage of the neurons to be covered. the `calculateStandardDeviation` method achieves this interpolation.

Next step is to retrieve new data from the `AngleReader`, which is done by calling `getData`. If any new angles were retrieved, then for each neuron in the network, we evaluate a normal distribution, with mean equal to the received angle and the standard deviation as calculated earlier. We make the incoming current for this neuron equal to the value. Once this has been done for each neuron, we call the `getSpikes` method on the neuron simulator, which simulates the network and provides us with any neuron spikes that have occurred. We add the retrieved spike pattern to the buffer and wait for a call to `step`.

4.9.2 Visual Input Channel

The `VisualInputChannel` class takes an image prepared by a `VisualReader` and attempts to convert it to a pattern of neuron spikes. The algorithm for this procedure has been described in the Design section of the Report. Before we can explain the workings of the `VisualInputChannel` class, we need to look at two related classes that a `VisualInputChannel` relies on.

If we recall, the image conversion process can be divided in two steps. The first step is the reduction of visual data or *foveation* stage, where we attempt to simulate the behaviour and distribution of the retinal photoreceptors. This stage is performed by a class called `LogPolarVisualDataReducer`, which

is a subclass of the abstract `VisualDataReducer`, which enables us to add other foveation algorithms to iSpike, some of which we have looked at earlier in the report. The second stage is the simulation of the colour opponent ganglion cells, which is performed by a class called `DOGVisualFilter`. This class extends the `VisualFilter` subclass, which in the future should allow us to add other Visual Filters, such as one that is based on light intensity as opposed to colour.

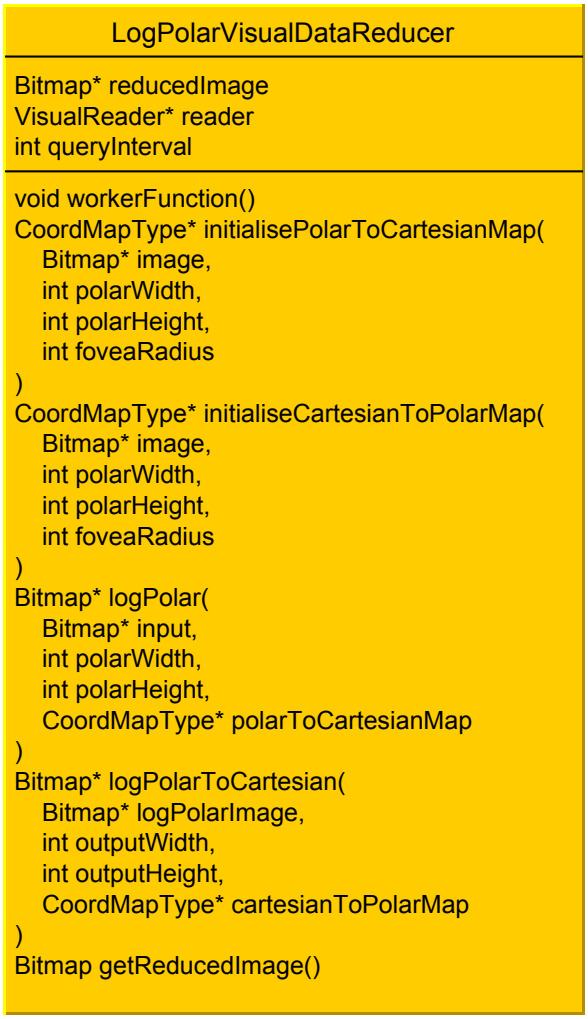


Figure 4.11: The `LogPolarVisualDataReducer` class

Figure 4.11 shows an overview of the `LogPolarVisualDataReducer` class. This class has four auxiliary methods. The `initialisePolarToCartesianMap` method, as the name suggests, initialises a mapping between polar and Cartesian coordinates, given an image, it's width and height, and the desired radius for the central fovea region. The `initialiseCartesianToPolarMap` method performs the reverse mapping from Cartesian to polar coordinates, given the same parameters. The `logPolar` method converts a given image from Cartesian to log-polar representation. `LogPolarToCartesian` is an auxiliary function that can be used to map the converted image back to Cartesian coordinates, this will preserve the foveation effect and can be used for testing and presentation purposes as a Cartesian representation of an image is easier to comprehend visually than a polar one.

The `LogPolarVisualDataReducer` class has it's own worker thread that continuously performs concurrency. We exploit concurrency here, as foveation can be a lengthy process and performing it in a concurrent fashion enables significant efficiency gains. The work loop of this class queries the associated `VisualReader`. If a new image is available, it is retrieved. Afterwards, the bidirectional coordinate mappings are created and the image is converted from Cartesian to log-polar coordinates. The log-polar representation is then added to the local buffer and the loop continues. The `queryInterval` variable controls the rate at which the Reader is queried. The produced log-polar image can be retrieved by calling the `getReducedImage` method.

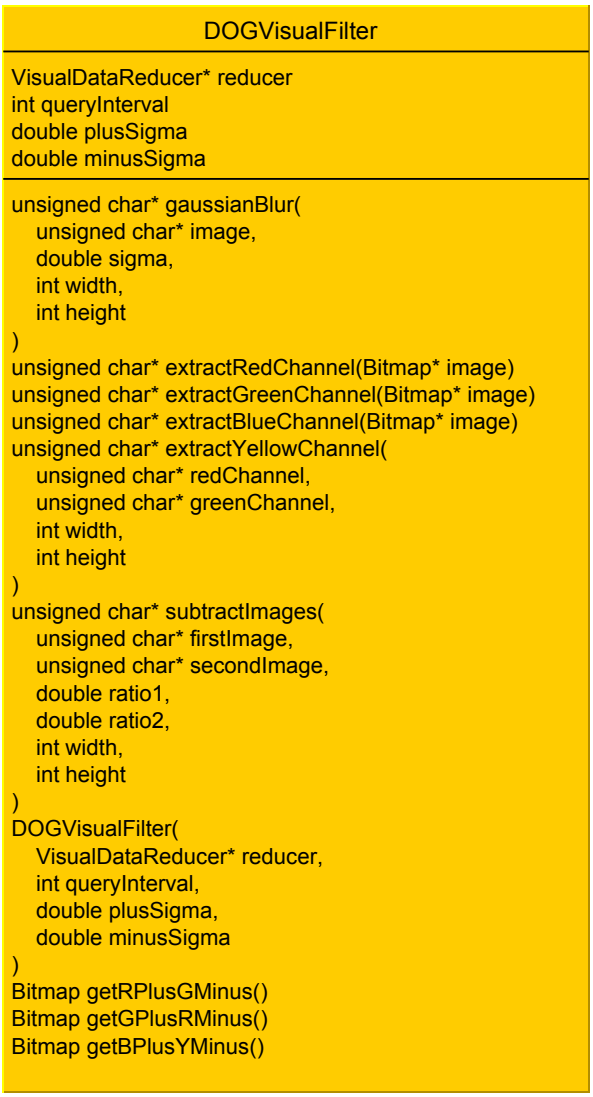


Figure 4.12: The `DOGVisualFilter` class

An overview of the `DOGVisualFilter` class is available in Figure 4.12. The `gaussianBlur` method performs applies a Gaussian blur filter to a given image. The sigma parameter controls the intensity of the blur, width and height provide the dimensions of the image. The next three methods extract the individual colour channel from the image, red, green or blue. The `extractYellowChannel` image

combines the outputs of `extractRedChannel` and `extractGreenChannel` to produce the yellow channel. Given two images, the `subtractImages` subtracts one from the other. The ratio parameters can influence the ratio between the two images.

One of the `DOGVisualFilter` fields is a `VisualDataReducer`, this is due to the fact that this function also has an internal processing thread to gain extra efficiency via concurrency. The execution loop of this class first retrieves a foveated image from the `VisualDataReducer` by calling `getReducedImage`. After that, the individual colour channels are extracted from the image and the yellow channel is created by combining the red and the green. Afterwards Gaussian blur is applied to the colour channels with varying intensities depending on whether the image is a minuend or a subtrahend in the following subtraction. The blurred channels are then subtracted from each other to form three colour opponent maps: *redPlusGreenMinus*, *redMinusGreenPlus* and *bluePlusYellowMinus*. These images can afterwards be retrieved by calling an appropriate *getter* function. The `queryInterval` field is, again, used to control the rate at which the loop executes.

Having explained the behaviour of the `LogPolarVisualDataReducer` and the `DOGVisualFilter` classes, the behaviour of the `VisualInputChannel` is relatively simple. During initialisation, this Channel starts an associated `DOGVisualFilter` and initialises a Neuron network. During runtime, the `VisualInputChannel` will retrieve an opponent map from the `DOGVisualFilter` indicated by the `opponentMap` parameter. After this, a current map is created, where each pixel maps to a neuron and the incoming current is proportional to the light intensity of that pixel. The current map is passed to the associated neuron simulation class and the resultant spike pattern is then retrieved. The spike pattern is added to the local buffer and the thread waits for a `step` command.

4.10 Output Channels

We use the concept of an *Output Channel* to indicate a process that takes a spike pattern as it's input, converts it into *a given format* and then uses a `Writer` to deliver them to a predefined destination. So, for example, a `JointOutputChannel` is a class, capable of converting neuron spike patterns into joint movement motor commands and then using an `AngleWriter` to deliver these.

4.10.1 Creating an Output Channel

All Output Channels are created via the `OutputChannelFactory` class. This class is similar to the `InputChannelFactory` class in structure and function. When an `OutputChannelFactory` is constructed, a vector of all available Output Channels is created. The user can then call the `getAllChannels` method to retrieve the all available channel descriptions. Having selected the desired Output Channel, a Writer of a supported type needs to be created via the `WriterFactory` class. The supported Writers for each Output Channel as well as the channel properties are available via an `OutputChannelDescription` Data Transfer Object.

Having created a Writer and initialised the properties of the new Output Channel, it can be created via the `create` method of the `OutputChannelFactory.`

4.10.2 Output Channel at runtime

Once created, the new Output Channel can be started by calling the `start` method. This will initialise the inner buffer, will start the associated Writer and will initiate the worker thread of the class. The worker thread normally contains a loop, which continuously inspects the local buffer to see if any new spike patterns have been added by the `setFiring` method. If this is the case, The first entry in the buffer is retrieved and converted into the output format. The converted data is then transferred to the associated Writer, which delivers it to a destination. The execution of the loop then halts and the Output Channel waits for a `step` command to continue the execution.

4.11 Supported Output Channels

4.11.1 Joint Output Channel

For the moment, the only Output Channel type supported is the `JointOutputChannel`. This is a class capable for receiving a spike pattern and converting it into an angle. This angle is then issued to an `AngleWriter` which, depending on the Writer, can store it in a file or convert it into a motor command and deliver it to YARP. The class has a number of parameters, the *Minimum Angle* and *Maximum Angle* parameters constrict the angular space to which the spike pattern is mapped. The *Rate of Decay* parameter controls the rate at which the value of the internal receptors decays over time. The *Neuron Width* and *Neuron Height* parameters provide the dimensions of the neuron network producing the incoming spike patterns.

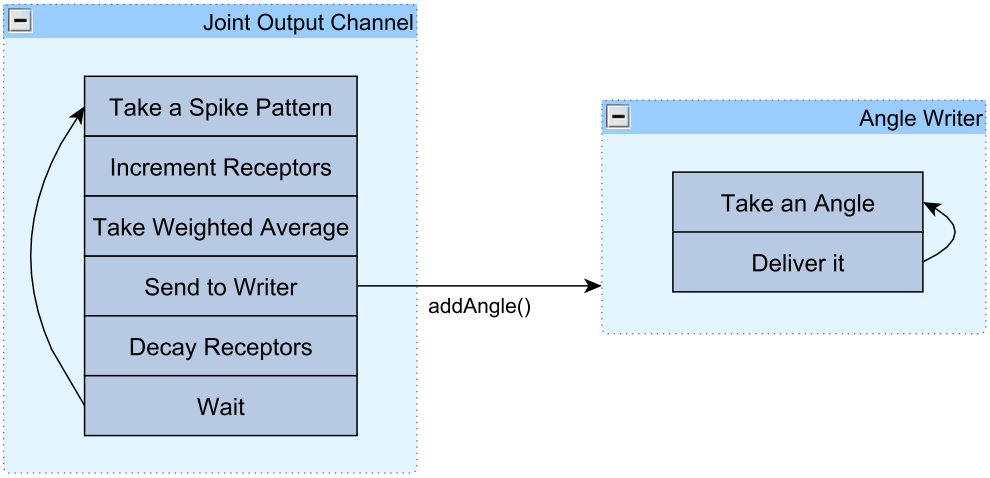


Figure 4.13: A `JointOutputChannel` along with an associated Writer

Once started, a `JointOutputChannel` will look at the spike pattern buffer, if the buffer is not empty, the Channel will iterate through the spike pattern and increment the value of those receptors, which have received a spike. We then produce a *weighted average* of the receptor values and this gives us a single angular value. This angle is then passed on to the associated `AngleWriter` via the `addAngle` method. We finally decay the value of every variable according to the decay parameter of the channel

and then the thread falls asleep, until a `step` command is given. Figure 4.13 shows the behaviour of a `JointOutputChannel` with an associated Writer.

4.12 Example Use Cases

Figure 4.14 shows an example usage of iSpike with two Channels working in parallel: a `VisualInputChannel` and a `JointOutputChannel`. The `VisualInputChannel` is used to retrieve visual information from a YARP port (with the help of a `YarpVisualReader`), convert it into a sequence of spike patterns for the SNN Simulator to use. The SNN Simulator produces an output spiking pattern which is given to the `JointOutputChannel`, which converts it to an angle and transfers it on to a `YarpAngleWriter`. The `YarpAngleWriter` converts the joint angle into a motor commands and delivers it to a YARP port. This combination of Channels could be used, for example, in conjunction with a neural network that uses the visual input from the iCub to guide and control it's locomotion.

Another useful Channel combination, that is supported at the moment, would be a `JointInputChannel` and a `JointOutputChannel` pair, working in parallel. This would normally be used with a neural network that takes a joint angle as it's input and gives a new target angle for that joint as it's output. The new angular value for the joint becomes a function of the current joint position. This enables, for example, state based progression of a particular joint through a number of predefined angles. If a number of `JointInputChannel`/`JointOutputChannel` pairs are used, a state based progression of parts of the robot, or the whole robot through a number of predefined states can be achieved.

4.13 Documentation and Unit Tests

For automatic documentation generation purposes, the *Doxygen* [36] toolkit is utilised with iSpike. Doxygen is a documentation system HTML or Latex format documentation based on comments written in the source code by the programmer. This removes the burden of maintaining a disjoint API documentation for iSpike and enables us to generate up-to-date documentation effortlessly.

We also introduced a number of unit tests with the help of the *Boost Unit Test Framework* [34]. The Boost Unit Test Framework is part of the Boost library set and provides a convenient way to create unit and acceptance tests for a given C++ project. We used the library to create a number of unit tests demonstrating the usage and behaviour of individual iSpike components. Having a good number of unit tests in a project is good, as the test set serves as additional documentation for any new programmer starting development on the project.

4.14 Console Client

A console based client for iSpike was developed as a part of the development process. The client serves as an example illustrating how to integrate with iSpike and demonstrates the Channel, Reader and Writer creation process and usage. Once started, the user can select, whether to create an Output or an Input Channel, afterwards a list of all available Channels of the given type is shown to the user. After selecting a Channel, the user is queried for parameter values for that particular Channel. Having provided all of the Channel parameters, a list of supported Readers or Writers becomes available. Once

the user selects an entry in the list and fills in the properties for the selected Reader or Writer, a new instance of the selected Channel is created, coupled with the selected Reader or Writer. In the case of an Input Channel, the spike patterns retrieved are displayed visually in the console prompt.

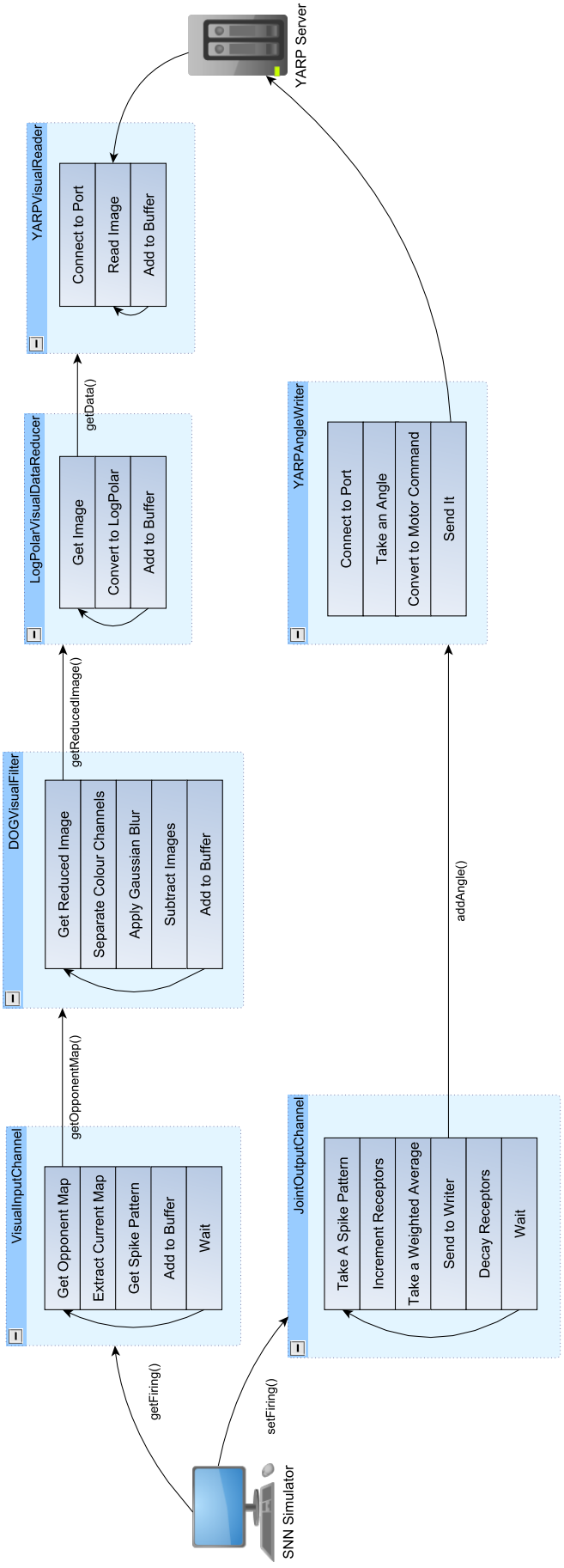


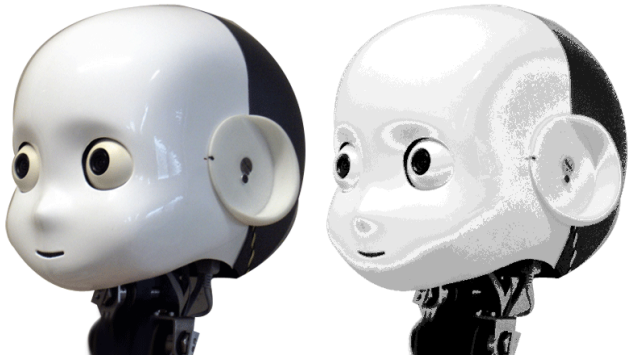
Figure 4.14: Example use of iSpike with a `VisualInputChannel` and a `JointOutputChannel`

5 Evaluation

In order for us to evaluate and measure the quality and fitness of our implementation of iSpike, we devised a number tests that would cover different aspects of the software.

5.1 Image Conversion Accuracy

We first tested the conversion accuracy of our `VisualInputChannel` implementation. We did this by constructing and starting a `VisualInputChannel` instance coupled with a `FileAngleReader` We disabled the foveation and opponency map generation components and used the intensity values of each pixel in the original image as current values for each neuron in the `IzhikevichNeuronSim` matrix. We then attempted to reconstruct the original image from the spike patterns by relating the intensity values of the new image to the spike time difference between the first and second spike for each neuron, essentially implementing a relative latency coding algorithm.



(a) Original image (b) Reconstructed image

Figure 5.1: Image conversion accuracy test results

Figure 5.1 shows a representative results of this experiment. On the left hand side is the original image and on the right hand side is the reconstruction. By visually examining the two images it can be seen that not much information has been lost by the conversion process and latency coding can be used to reconstruct the original image to a high degree of accuracy.

5.2 Joint Angle Conversion Accuracy

5.2.1 Experimental Design

Firstly, we wanted to see how accurate our implementation of the conversion processes is. The most straightforward test we could do is use the fact that we can convert joint angles into spike patterns and

then convert a spike pattern back into a joint angle. By creating an instance of a `JointInputChannel` in parallel with an instance of a `JointOutputChannel` we can encode a given angle into a spike pattern and back into an angle. Then by comparing the difference between the two angles we get an encoding error value. The only cause of this error would could be loss of precision or inaccuracy in either of the conversion processes.

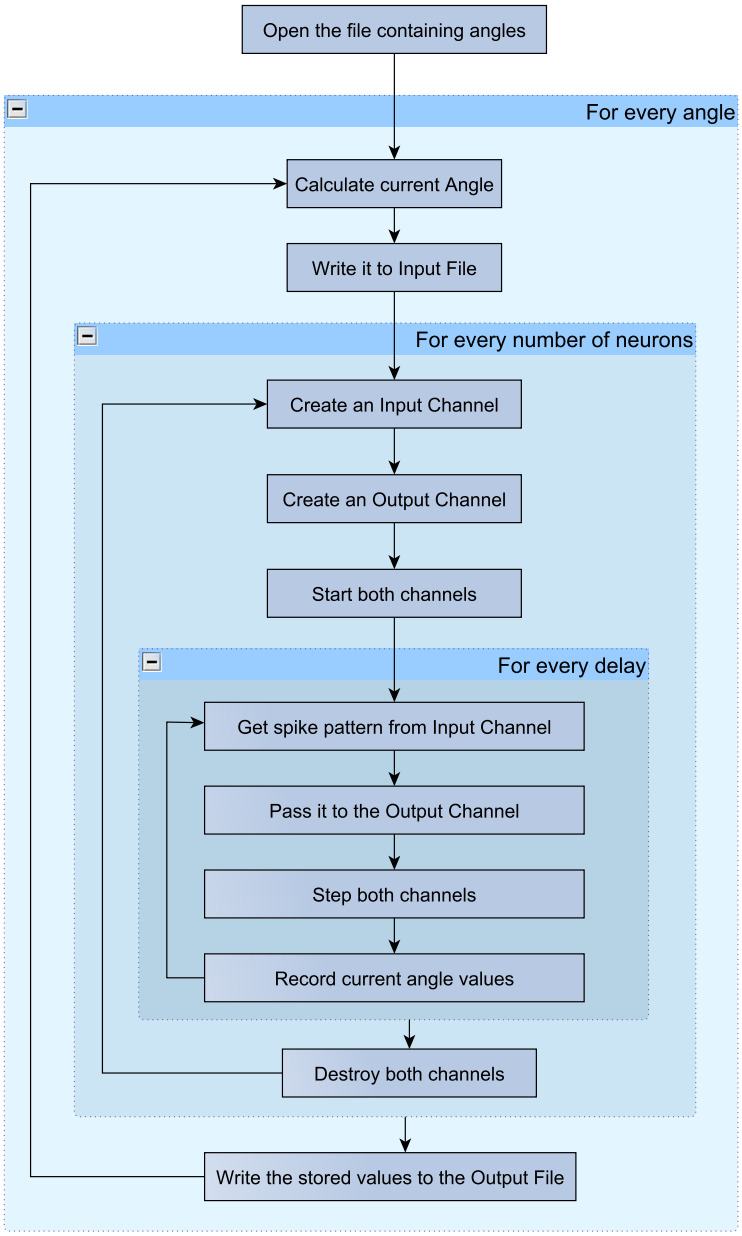


Figure 5.2: The Conversion Accuracy Experiment

We implemented this test in the form of a unit test, as it uses a large part of the functionality offered by iSpike and hence can be useful for testing purposes. Two files are used by the unit test, the input file contains a list of angles we would like to encode. It is useful to attempt to encode a variety of angular values, as this will highlight any issues with encoding particular angle values, if

there are any. The output file will contain the results of the test. We use a file to make it easier to import the data into a software that would allow us to evaluate the results, such as Matlab. This file contains five columns: the *original angle*, the *number of neurons used*, the *length of the current run*, the *current angle of received by the Input Channel* and the *current angle of the Output Channel*. The Input Channel angle should always equal the original angle and to get a measure for the accuracy we compare the current angle of the Input Channel to the current angle of the Output Channel, we would like these to be the same.

The experiment is structured so that first we process the contents of the input file. Then we iteratively increase the number of neurons used by both channels and create an instance of both channels. We use a `FileAngleReader` and a `FileAngleWriter` to enable reading the input angle from a file. Having created the two channels, we start them. At this point, the Input Channel will read the next angular value from a prepared file and will attempt to encode it.

At this point we have a loop, where we continuously step both Channels and provide the Output Channel with the spike pattern produced by the Input Channel. The spike pattern represents the original angle, so the Output Channel should convert it back to the original value. We then record the both angular values and add an entry to the input file. Note that in this loop, the angle being encoded and the number of neurons is fixed. The only variable is the number of `step` commands we give. We would expect the encoding precision to increase with the number of iterations, as the Output Channel receives more spike patterns and the receptor values converge to the actual angle over time.

Once we have processed every number of neurons, we store the collected values in the output file, take the next angular value and continue until done. The workflow of this experiment is shown in Figure 5.2.

5.2.2 Experimental Runs

By varying the number of angles, the number of neurons and the length of the experiment we can define three separate experimental runs, each evaluating a specific aspect of the coding process.

Individual angle conversion accuracy

In this case we fill the input file with a large number of angles and run the experiment with a fixed number of neurons (30) and keep the delay fixed as well(100), we can observe if there is any impact on the conversion accuracy by the value of the angle itself. We filled the input file with more than 900 randomly generated angle values between 90 and -90 degrees and ran the experiment.

Figure 5.9 shows the results of the experiment. The top graph has two curves, the blue curve represents the input angles and the green curve represents the output angles. On the Y axis we have the different angular values and on the X axis we have time. It is easy to see that the two curves almost completely overlap with a few exceptions, which overall seems to be a good results and indicates a close correspondence between the input and output angles.

The middle graph uses the same axes, but has a single blue curve, which is the difference between the input and output angles, or the error. Ideally this curve would be equal to zero throughout the experiment, which would indicate no error in the conversion process. By looking at the actual curve this is not quite true, but the curve is very close to zero throughout the experiment except for a limited

number of outliers. The outliers are evenly spread through the angular space, which indicates that the error is not dependent on specific angle or range of angles.

The third graph shows the plot of Input Angles versus the Output Angles. Here we would like to see a direct proportionality between the two axes shown as a straight diagonal line. As we can see, this is mostly true apart from the small number of outliers seen in the previous graph.

Impact of the neuron count on the conversion accuracy

This run essentially consists of repeated runs of the previous type with varying number of neurons used for conversion. Instead of looking at the individual conversion error for each angle, we aggregate the results and produce a root mean squared error value for each run by calculating the mean error for a given number of neurons and then taking the square root of this value. This enables us to see what impact the number of neurons we use has on the conversion accuracy.

Top part of Figure 5.10 shows the results obtained. The graph shows that by increasing the number of neurons used up to 10 we quickly improve the conversion accuracy. If we further increase the neuron count, the accuracy does not seem to improve and instead, starts to decrease slowly. The graph seems to indicate that using ten neurons is close to optimal for joint angle encoding purposes. During this experimental run, we give a fixed time for the Output Channel receptor field values to converge. We think that the gradual decrease of the conversion accuracy seen if we continue to increase the number of neurons past ten can be explained by the inability of the receptor values to converge, given the time constraints, as demonstrated by the next experimental run.

Impact of the convergence duration on the accuracy

As mentioned in the Design section of the report, `JointOutputChannel` contains a number of receptor values, that increase with incoming spikes and decay over time. The weighted average of these receptor values will converge to the actual joint angle value over time. We would like to evaluate the convergence dynamics for varying numbers of receptors. For this, we devised an experiment, where we pick a fixed number of neurons and receptive fields. We then encode and decode a large number of angular values and record the difference for each time step of the inner loop. By plotting the RMSE value, for a given number of neurons against the time duration of the experiment we get the bottom graph of Figure 5.10.

There are three curves on the graph, the blue curve shows the results for 10 receptors/neurons. The green curve shows the results for 30, the red curve for 90. An immediate observation we can make, is that when using a larger number of neurons/receptors, the conversion accuracy takes longer to improve, then by using a smaller number of neurons. This can be explained by the fact, that for high numbers of neurons and receptors, the spike patterns will be "richer" and more complicated, as more neurons will receive a high enough current to fire. If only a single neuron fires in results to an incoming angle, the weighted average of the receptor values will give an instant value and will not change over time. In effect, it converges instantly. The more neurons spike, the longer it would take for the incoming spike rate and the decay rate to find balance, which leads to a longer time for the weighter average to converge to a single, stable, value.

A less expected observation is that for smaller neuron numbers, the curve seems to take longer to

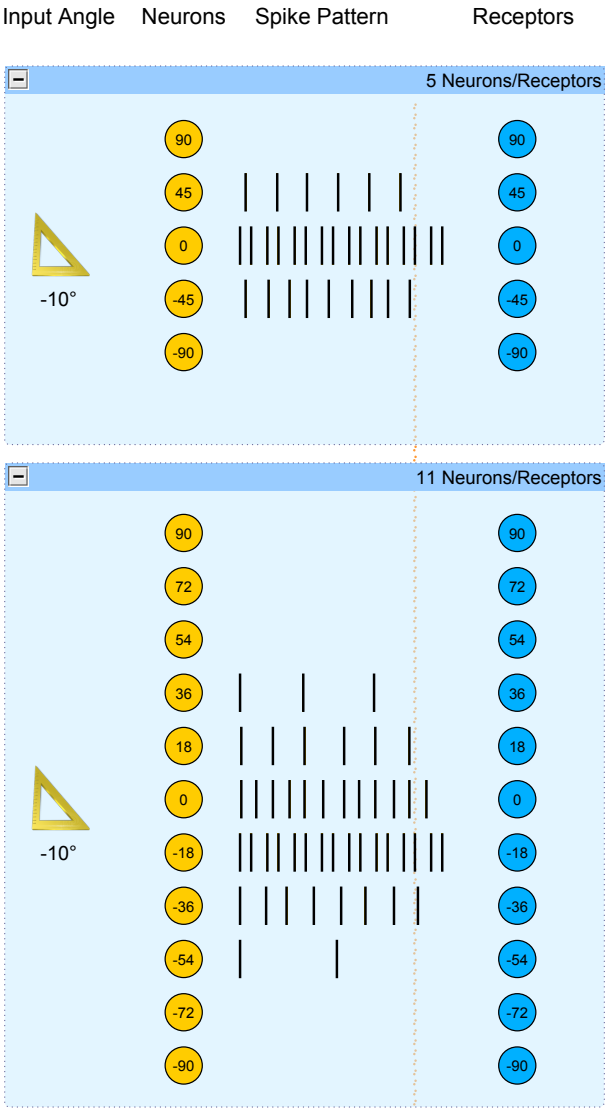


Figure 5.3: An illustration of difference in convergence for a smaller and a larger number of neurons / receptors.

start "coming down", i.e. it takes longer for the RMSE values to start do decrease. We think this can also be explained by the more information rich spike patterns when a larger number of neurons and receptors is utilised. When using less neurons, there will be less neurons producing spikes for a given angle, this will lead to fewer receptive fields having differing values, with the extreme case of one neuron firing leading to a single differing receptor value, which might be far away from the actual angular value. In contrast, when a large number of neurons spike, the first few spike patterns will produce a higher variance in the receptive field values leading to a more accurate initial approximation. This approximation will be better early on, but will take longer to converge to the actual angular value, as it takes longer for the decay rate to differentiate between neurons spiking at a similar rate.

Figure 5.3 visually demonstrates the impact of the number of neurons on the produced spike pattern. The figure shows two sets of neurons and corresponding receptors. The top set has five neuron/receptor

pairs, the bottom set has eleven. In both cases the input angle is -10 degrees. The dotted vertical line crossing the figure shows the difference between spikes received in the first few moments. By the time the first spikes from two neurons have arrived, the bottom receptor set has received spikes four neurons. This is caused by the fact that two out of the three neurons that produce spikes in the top figure have less incoming current and hence take longer to produce a spike. The accuracy of the weighted average value of the receptors is proportional to the number of receptors that have received spikes, hence the eleven pair set will produce a better average value early on. It will however take longer for this value to *stabilise* or converge. This is due to the fact that, for example, the 0 and -18 degree neurons have similar firing rates and can not be easily distinguished. It will take more time for the decay in receptor values to differentiate between the two.

5.3 Speed Of Execution

We also wanted to see how fast our implementation of the conversion processes is and how close we are to achieving real time conversion. For this, created an experiment where we test each component of iSpike separately, to see the impact each component has on the overall performance. The components we decided to test are the *Neuron Simulator*, the *Visual Data Reducer*, the *DOG Visual Filter*, and all the Channels. We decided not to test separately the Readers and Writers as their impact on the performance is likely to be negligible due to the simplicity of their behaviour. We would expect the major bottleneck to be the Visual Data Reducer and the DOG Visual Filter components, due to the complex processing on potentially large images carried out by these components.

All tests were done on a single desktop PC with an Intel Core 2 Quad Q6600 CPU, that has 4 cores, each at 2.4 GHz. The PC had 4 GB of DDR2 RAM and was running a Windows 7 64 bit operating system.

5.3.1 The Izhikevich Neuron Simulator

Our first experiment was done to see how efficient our implementation of the Izhikevich neuron simulator is. For this, we created a unit test that created an instance of the simulator with increasing numbers of neurons. The networks were then simulated for 10 seconds and the number of spike patterns retrieved was recorded. By increasing the network size, we could see how the efficiency is related to the size of the neuron network, which might let us put an upper limit on the number of neurons that can be used in a real time application. The test was repeated 10 times and the average number of spikes patterns per second was plotted against the number of neurons in the network.

Figure 5.4 shows the test results. For a network consisting of 1000 neurons, we managed to produce close to 10000 spike patterns per second. The curve seems to exponentially decrease with 1000000 neurons only giving on average ten patterns per second. These results are encouraging, as from earlier experiments it seems one would rarely want to use more than 100 neurons for encoding joints. In regards to visual information, we have one neuron per pixel in the opponency map. The picture resolution that the iCub uses is 320x240, which gives us 76800 neurons. According to the figure, we should be able to produce around 300 spike patterns per second at this resolution, which is a good result, considering the visual feed of the robot is only 15 frames per second.

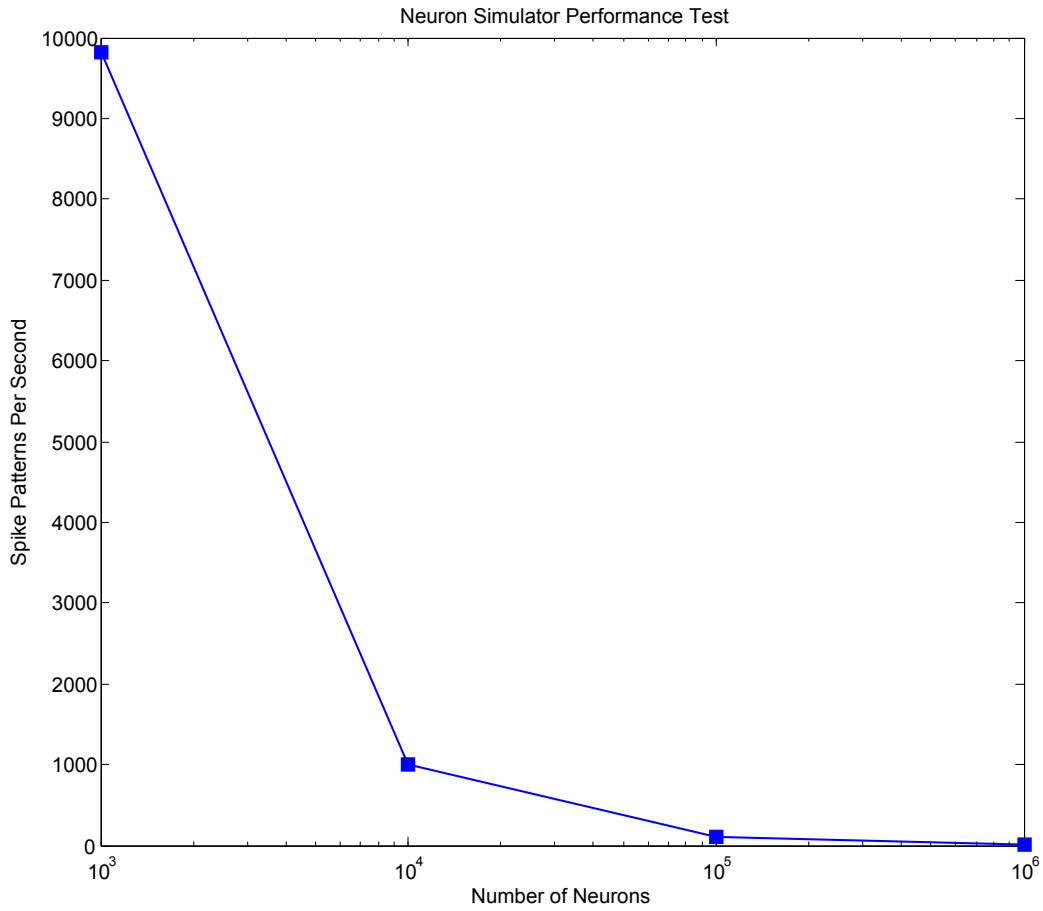


Figure 5.4: Neuron simulator performance test results

5.3.2 The Visual Data Reducer

We were interested in the runtime performance of our Visual Data Reducer. To measure this, we created a unit test that creates and initialises a `LogPolarVisualDataReducer` and foveates a given image. We let the reducer run for 10 seconds and count how many images it has foveated. We do this for increasing image sizes to see what effect the image resolution has on the number of frames that can be foveated. The performance figure retrieved also include any overhead caused by the Reader, but we expect this to be negligible, as the images are prepared before the test and passed to the Reader via local memory.

Figure 5.5 shows the results of this experiment. The image resolution in each dimension (such that 500 means an image resolution of 500x500) is shown on the X axis and the number of images processed by the Visual Data Reducer is shown on the Y axis. Note, the Y axis are logarithmically labelled. The curve, again, is declining exponentially with close to 115 images per second produced using an image of 50x50 resolution. It takes 3 seconds to produce a single image with a resolution of 1000x1000. According to the curve, the Visual Data Reducer can produce four to five images per second at the resolution used by the iCub, which is three to four times less than the frame rate used by the camera. Because of this, currently it is impossible to achieve real time foveation with the current Visual Data Reducer.

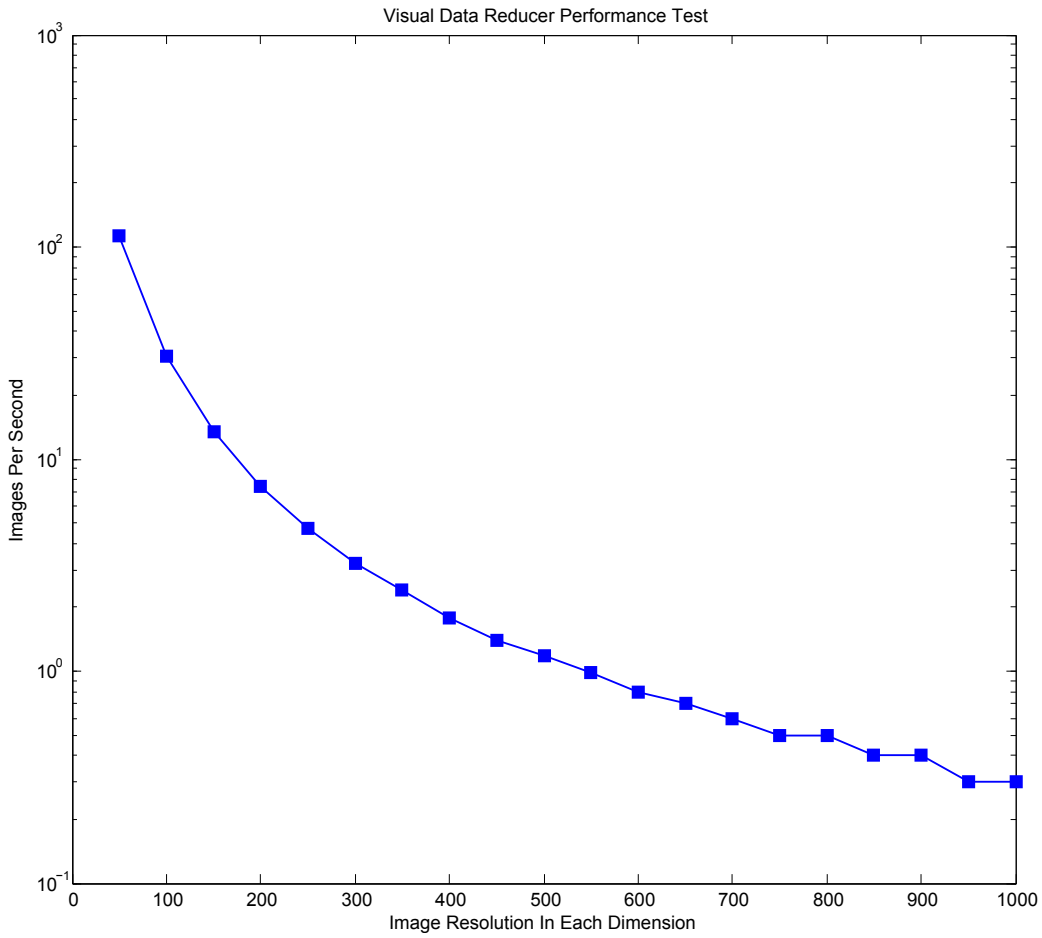


Figure 5.5: Visual Data Reducer performance test results

5.3.3 The DOG Visual Filter

This test is similar to the previous, except this time we are interested in the runtime performance of the Difference-of-Gaussians Visual Filter. Especially, due to the fact that it is the most computationally intensive component of iSpike and is likely to be the bottleneck in any given use case. For this test, we created an instance of a `DOGVisualFilter`, initialised and started it. We then let it execute for 10 seconds and counter how many images were processed during this time. We repeated this test for increasing dimensions of the incoming image to observer the relationship between the image size and the time taken to extract the opponency map of that image.

Figure 5.6 shows the results of this test. We can see that the shape of the curve is similar to that obtained in the Visual Data Reducer test. The curve seems to be exponentially decaying with the size of the image with a 50x50 image producing 14.7 opponency maps per second and an image of 500x500 only producing a single opponency map every five seconds. For the resolution used by the iCub, the Visual Filter is only capable of outputting one opponency map every 1.5 seconds. This result is to be expected as applying a Gaussian convolution is a computationally expensive task. This rate is much slower than the 15 frame per second rate of the camera used by the iCub, hence at the moment the opponency map is not updated as quickly as the visual view might be. Nonetheless the current rate is arguably fast enough for use in scenes that do not contain rapid movements or events.

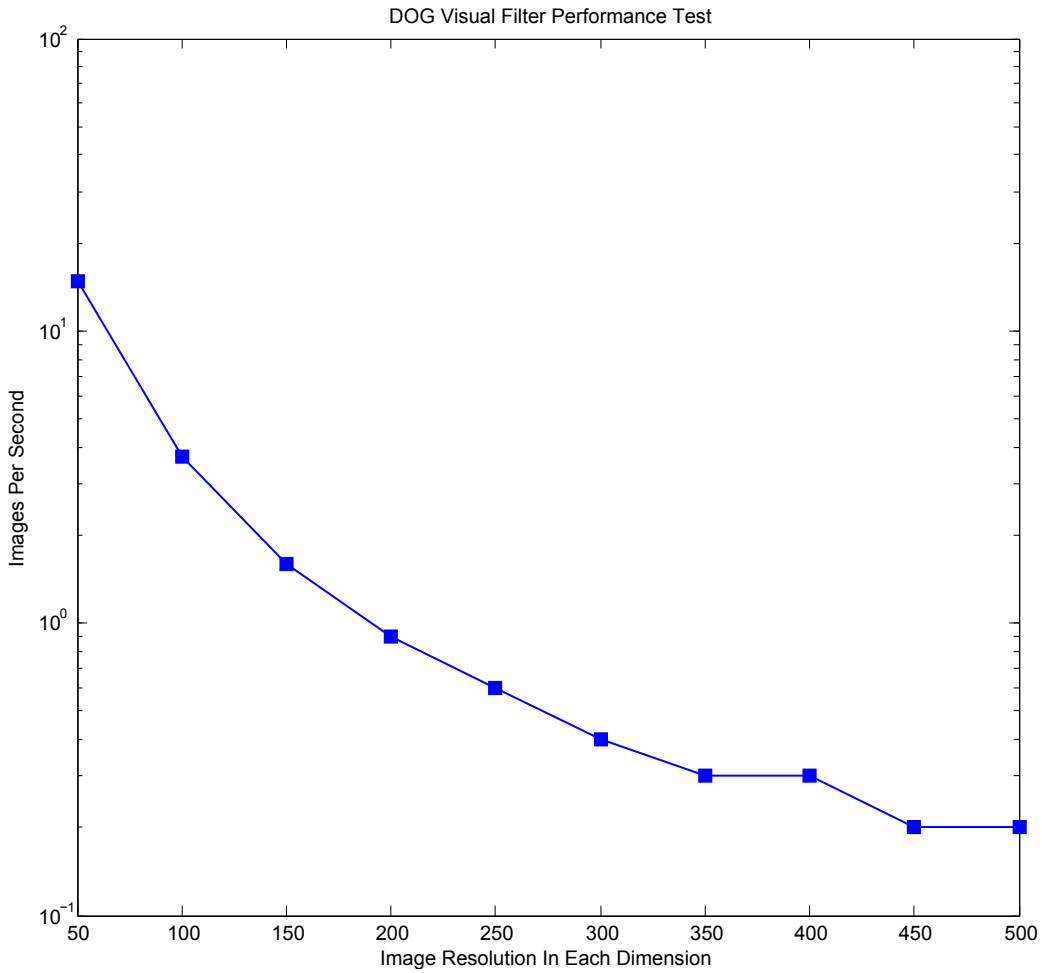


Figure 5.6: DOG Visual Filter performance test results

5.3.4 The Channels

We then proceeded to test the performance of the Input and Output channels to evaluate the effective rate at which information can be retrieved and delivered by iSpike.

We started by testing the `JointInputChannel`, for this task we created an instance of a `JointInputChannel` coupled with a `FileAngleWriter`. We appended a single entry to the input file for the reader, let the Input Channel run for 10 seconds and counted the number of angles retrieved by the Channel. As the `JointInputChannel` is relatively simple and does not perform any computationally expensive tasks, the retrieved rates were very high and on average the Channel produced 180424 joint angles per second, which demonstrates that the `JointInputChannel` can be used for real time retrieval and processing of incoming joint angles.

We then tested the `VisualInputChannel` in a similar fashion, by creating and initialising an instance coupled with a `FileVisualReader`. The Channel was let to run for 10 seconds and the number of spike patterns per second produced was recorded. The retrieved value was unexpectedly high (870 frames per second on average) and did not seem to be impacted in a major way by the size of the image. This is probably due to the fact that, while the underlying conversion algorithms are computationally expensive, a caching mechanism is used by both the `DOGVisualFilter` and the

`LogpolarVisualDataReducer`, which stores the last converted image in a buffer and makes it available upon request until a new image is produced. This ensure very high retrieval rates even if the associated image conversion rate is relatively low.

We also tested the `JointOutputChannel` by first creating an instance of one, coupled with a `FileAngleWriter` and then feeding it a constant firing pattern for 10 seconds. We counted the number of individual joint angles produced by the channel during this period giving us a measure of produced angles per second. The value for the rate was, as expected, very high (on average 48412 produced angles per second). This is to be expected, as the Output Channel performs very limited tasks itself and the performance is mostly bound by the Neuron Simulator, the performance measure of which is shown earlier.

5.4 Simulation Test

Having done some accuracy and performance benchmarking, we wanted to create a working demonstration of iSpike, Spikestream/NeMo and the iCub simulator software. This would prove that iSpike can be used for it's initial purpose, which is to facilitate communication between the iCub and the SNN simulator. For this test we had to implement a wrapper for iSpike in the Spikestream software, which would make the iSpike functionality available from the Spikestream GUI. This was done by the author of Spikestream – David Gamez. David created an interface that allowed channel creation and execution at runtime from the GUI. He also implemented a mapping on the Spikestream side that maps the spike patterns produced by iSpike to the neurons in the simulator.

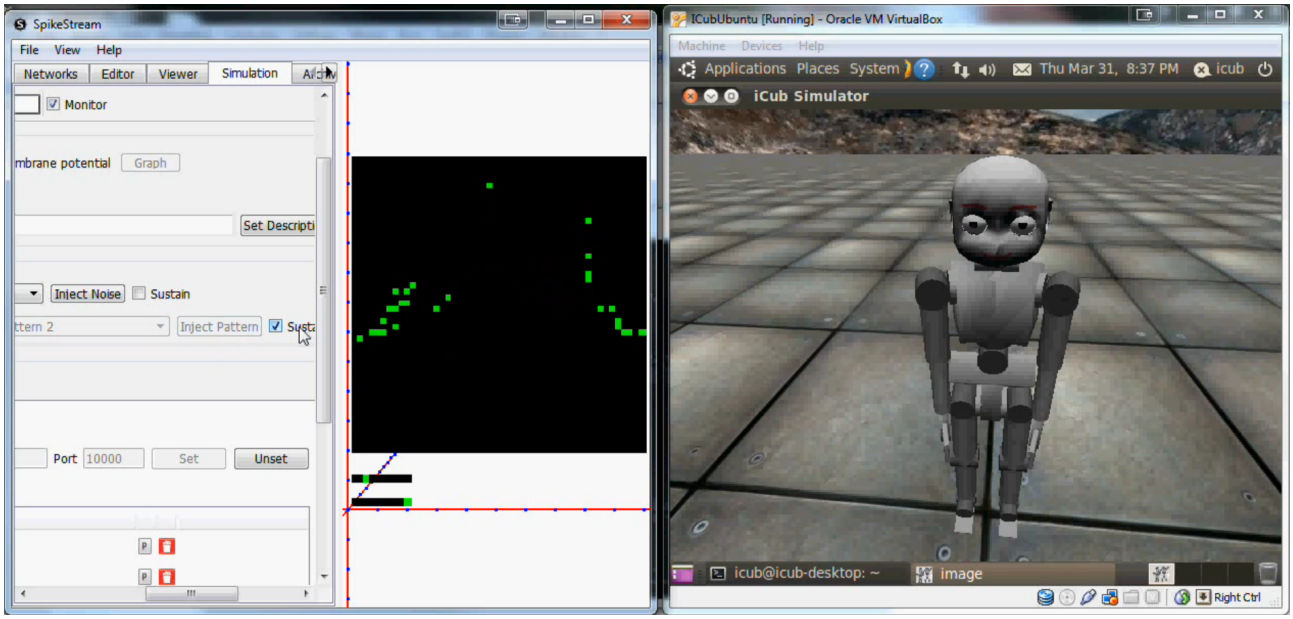


Figure 5.7: Single frame from the Simulation Test video

With such a wrapper in place, we created a demonstration simulation that used a `JointInputChannel`, a `VisualInputChannel` and a JointOutputChannel in parallel. The spike patterns produced by the Input Channels were mapped to two separate neuron groups. This allowed us to visually see the spike

patterns in the simulator as they arrive from iSpike. A separate array of neurons was mapped as the input to the Output Channel, which in turn sent motor commands to the iCub simulator head joint, essentially allowing the control of the virtual iCub's head from within Spikestream. A diagram demonstrating the experimental environment is available in Figure 5.8. The demonstration was captured on video and a frame from this video is available in Figure 5.7.

The figure shows the Spikestream GUI on the left and the iCub simulator on the right. In the GUI, the large, mostly black, rectangular section is the neuron group mapped to the visual input received from the virtual iCub. The neuron spiking pattern changes with the visual input from the iCub. There are two smaller arrays of neurons under the visual neuron group. The top array shows is mapped to the current position of the iCub head joint and the single spiking neuron indicates the angle of the joint. The lower array of neurons is mapped to the Output Channel, which sends motor commands to the head joint depending on which neuron is activated. In the picture, the rightmost neuron is activated and hence the head of the virtual iCub moves as high up as possible.

During the test we managed to control the movement of the head joint only by activating individual neurons in the relevant neuron group, from within Spikestream. The neuron groups mapped to the inputs of the iCub simulator correctly updated as the joint position and the visual view of the virtual robot was updating. For these reasons we conclude that the test was a success as we managed to create a working feedback loop between Spikestream and the iCub simulator.

5.5 Simulation Test With The Actual iCub Robot

A simulation test for the actual iCub robot is being designed at the moment and we are planning to perform it on the 24th of June using the iCub robot available at the University of Plymouth.

The planned experiment will consist of two phases, the initial phase will be an extended version of the Simulation Test performed on the virtual robot and will cover the control of multiple joints of the iCub robot simultaneously. We plane to have multiple neuron groups, each mapped to an Output Channel connected to a joint on the robot, enabling the control of each joint separately, or in parallel by injecting a spike pattern using the Spikestream GUI. A separate set of neuron groups will be mapped to Input Channels, each connected to the respective Joint angle sensor, allowing for real time monitoring of each joint from Spikestream. As before, a separate rectangular neuron group will be mapped to the visual input from the robot enabling us to observe the changes in the view from the iCub's eye as the joints are being moved.

The next phase of the experiment will consist of passing the sensory patterns received in the initial phase to a predefined neural network. The goal of this network will be to memorise a sequence of joint positions using Spike Time Dependent Plasticity (STDP) and to recall them on command, sending the appropriate motor commands to each joint so as to move the body of the iCub in the same position that was memorised earlier. The observed behaviour of the robot should be that initially we manually move the joints of the iCub through a number of "poses" and at the end of the experiment the sequence of these poses is "replayed" upon demand with no manual intervention from us.

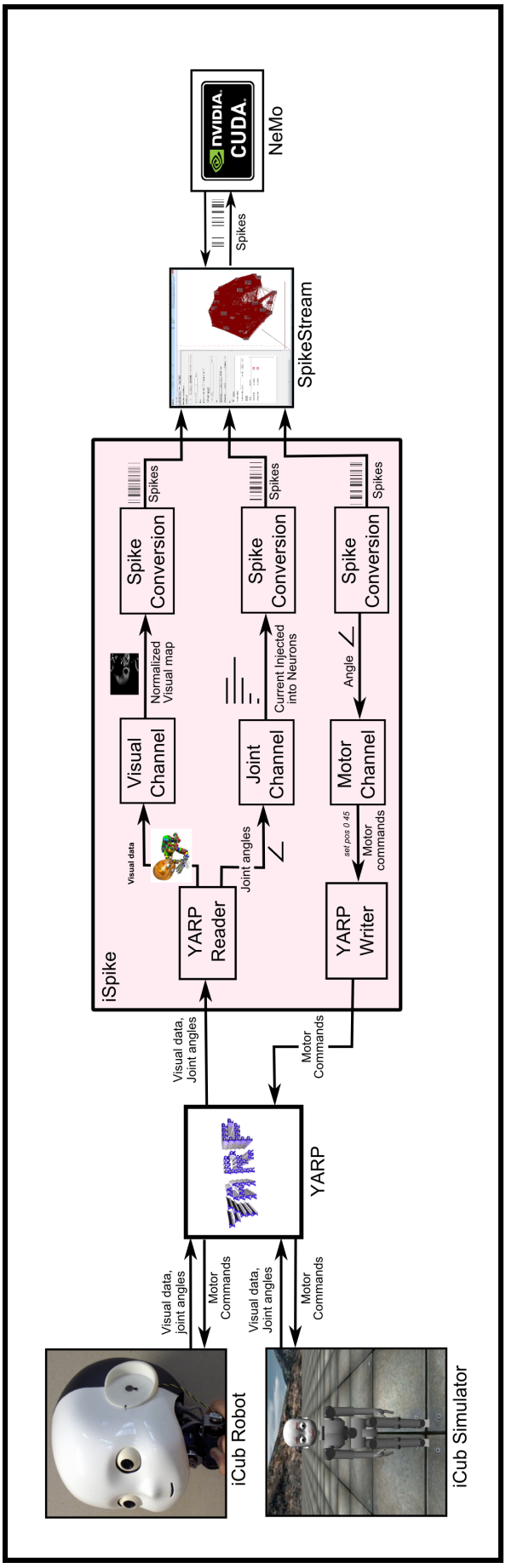


Figure 5.8: The experimental environment of the Simulation Test

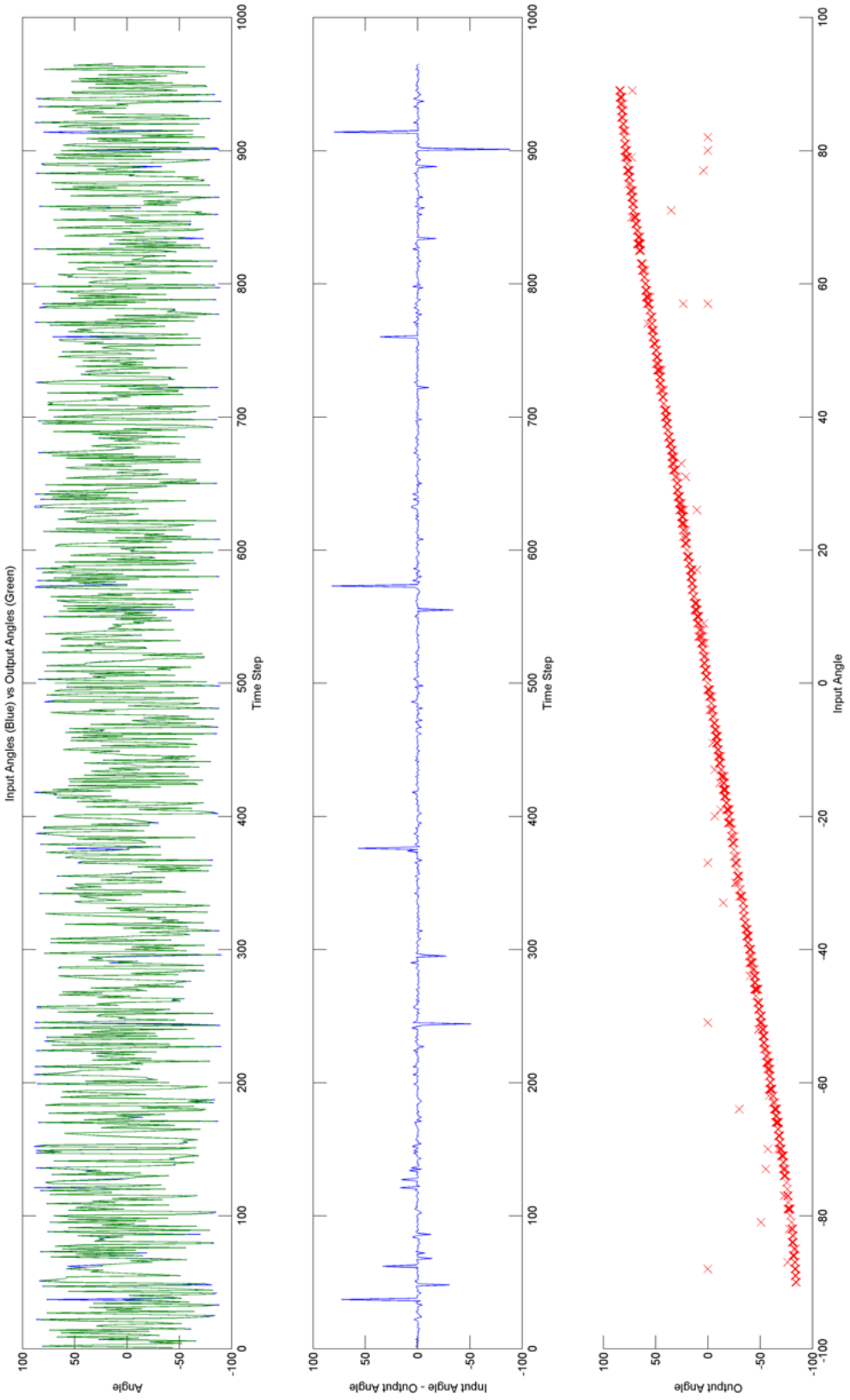


Figure 5.9: Experimental results with fixed number of neurons and a fixed convergence duration for each angle

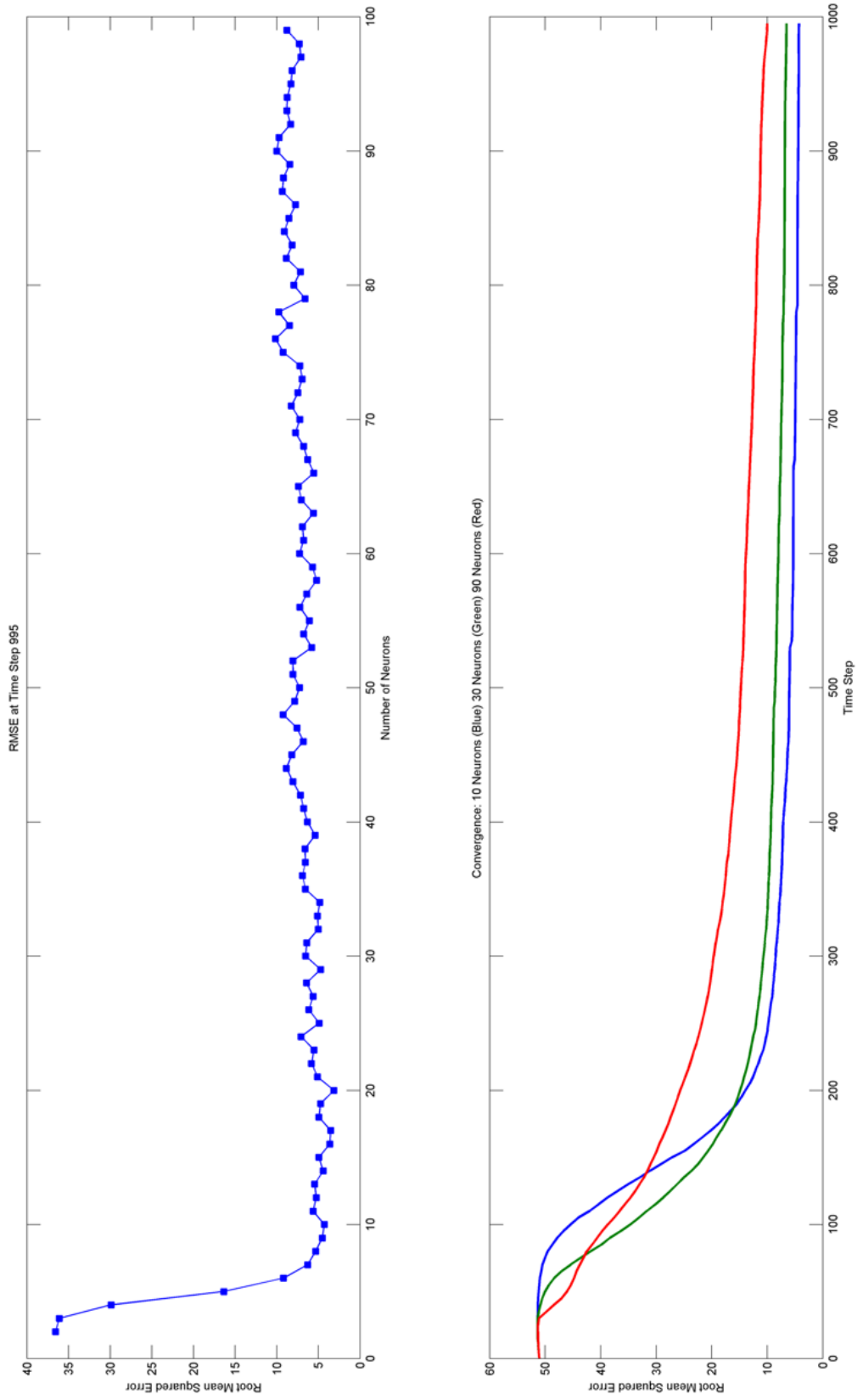


Figure 5.10: Top: Impact of the number of neurons used on the conversion accuracy Bottom: Convergence metrics for different numbers of neurons used

6 Conclusion

6.1 Summary

In conclusion, with this project we aimed to solve the problem of enabling bidirectional communication between a spiking neural network simulator and the iCub robot. With the development and successful testing of iSpike, we have proved the possibility of solving this problem and have provided one solution to it. We have used iSpike to control the virtual iCub robot from within the Spikestream/NeMo SNN simulator.

During the design and development phase, the complexity of the task quickly became apparent due to a variety of reasons. One of the main challenges has been trying to realise a biologically realistic conversion process for biological pathways that we only have a limited functional knowledge about, for example the motor control pathways of the spinal cord. Even in processes that are well researched, such as human vision system, applying the developed models in a fast performance environment is not a trivial task.

The scale of the project has proved to be another challenge. During the design and development phases we had to be very selective of the feature set to include in the initial build of iSpike, such as deciding which types of information to support, which models for neural networks, which models for each of the types of information. We had to decide how closely we would like to mimic the biological counterparts of each individual component.

Nonetheless, having solved the initial problem, even at a basic scale, the potential utility of having a feedback loop between a robot and a SNN simulator has quickly become apparent. By having a real time translation in place between the two sides, more high level computation and control can be achieved by the SNN simulator, which is not fixed in place, but instead can be dynamic and adaptable to the environment with the help of the information provided by the sensors on the robot and mechanisms such as Spike Time Dependent Plasticity.

6.2 Achievements

- During the development is iSpike we have shown that creating a bidirectional communication interface between neural network simulators and communication enabled robots is not only a viable pursuit, but is a very much achievable goal, as shown by our implementation and the Simulation Test performed as a part of the Evaluation process.

- As a part of the project, iSpike was released as a self contained open source library available at `http://ispike.sf.net`. Integration with the Spikestream/NeMo Spiking Neural Network Simulator and the iCub robot was achieved and the functionality was shown with a number

of tests and a working demonstration involving the control of the the virtual iCub robot from within Spikestream.

- In February 2011 a 3 page abstract application was submitted to the International workshop on bio–inspired robots which was to occur on the 15th of July, 2011. The submission, named *iSpike: A Spiking Neural Interface for the iCub Robot*, detailed the development of iSpike and the conversion processes we had implemented at that point. Our submission was accepted for an oral and poster presentation. We attended the workshop in July and presented our project during the Neural Processing session of the workshop. During the workshop, we received positive feedback from the attendants, increased public awareness on iSpike and promoted the use of iSpike as a general purpose communication interface for use with SNN simulation tools and actual robots, or their simulators. This was especially relevant to the attendants of this workshop as it concentrated on bio mimicry and the simulation of biological processes and phenomena in robotics. The fact that we attempt to be biologically plausible in our conversion components was well received by the participants and a number of guests showed interest in the use of iSpike in their own projects.

  Of particular interest was a discussion with Tim Lindgraf, a researcher from the Freie Universitt Berlin, who was interested in studying and emulating the behaviour and cognition of honeybees and the "dances" performed by the honeybees to communicate environmental information to other honeybees. Tim was planning to use a spiking neuron network resembling in structure and size a part of the honeybee brain to store, retrieve and represent location information for navigation purposes.

- There is currently an article in progress for publication in a special issue of the *Bioinspiration & Biomimetics* journal, which is dedicated to the submissions made for the workshop mentioned earlier. The deadline for this submission is 15th of July, 2011. The article will take form of an expanded version of the original submission, publishing our conversion accuracy and performance test results.

6.3 Future Work

As was mentioned earlier, during the design and development of iSpike, we have had to limit ourselves to a relatively basic feature set and instead concentrate on the core architecture due to the strict time constraints. As a result of this, there is a large number of possible extensions and improvements that can and should be made to iSpike, some of which we will detail in this section.

It would be very beneficial to achieve real time performance with iSpike in such a way that the processing of visual and joint angle information can be achieved at a higher rate than that of the sensor feeds of the iCub robot. Our main concern is with the implementation of the visual pathways and the Visual Channels. In their current form, the foveation and opponency map components are not up to standard in terms of efficiency as shown in the Evaluation section of the report. Both achieve a lower number of processed images per second than the visual feed of the iCub robot, which is 15 frames per second. There are a number of possible ways to improve the performance of both components.

Both the foveation and opponency map components of iSpike deal with visual data and as such, would benefit greatly from involving the Graphics Processing Unit (GPU) in at least a part of the processing. One option would be to use OpenGL(the Open Graphics Library) and implement the log polar and difference-of-gaussians transformation as convolution shaders. This could potentially exploit the advantages current generation GPUs have in terms of texture and image processing speed and parallelizability. This step alone should have a significant enough effect on the performance of the visual channels to bring the idea of real time performance much closer to realisation.

As shown by our performance tests, the Joint channels, on the other hand, are fast enough to be reliably used in a real time application with reasonable precision in terms of the number of neurons used.

While during the Design and Implementation phases we have made the core architecture of iSpike flexible and easy to extend, we have only provided a single working implementation of each channel and we use a single model for the internal neuron network. It would be worth considering other models, for example, in the Visual Input Channel. As discussed in the Background section of the report, there is a significant number of models that can be used in the foveation stage, some of which are more biologically plausible than the *log(z)* model we use in our implementation. Experimenting with other neuron coding methods would also be a worthwhile task, especially if we compare the performance and accuracy of other methods with the implementation we have at the moment.

We have also been limited in the number of sources and types of sensory data that are supported by iSpike as well as the possible outputs for motor commands. For example, in addition to the visual and joint sensors, the iCub also provides a pair of audio feeds and will soon be fitted with tactile sensors. It would be very useful to implement support for these and other types of sensory data. The structure of the Channels would be similar to the ones we have in place already, the difference would be in the method and algorithm used for conversion, which should be biologically plausible.

While being able to communicate with and transfer data to and from YARP has potentially made iSpike compatible with any robots that use YARP for communication, it would be useful to implement a number of Readers and Writers that would enable communication with robots that are not relying on YARP. The implementation of these would probably be specific to each individual robot.

In fact, the data received by Readers does not have to come from a sensory feed of a robot. It could potentially come from anywhere. For example, a Channel could be implemented that receives current stock prices from a financial service and uses a sensible encoding to convert these into a neuron spike pattern. Then the SNN simulator could be used to make evolving high level calculation with the gathered data.

Bibliography

[1] `http://eris.liralab.it/yarpdoc/note_ports.html`.

[2] E. D. Adrian. The impulses produced by sensory nerve endings: Part i.

[3] Mark F. Bear, Barry W. Connors, and Michael A. Paradiso. *Neuroscience: Exploring the Brain*. Lippincott Williams & Wilkins, 2007.

[4] Daniel Bendor and Xiaoqin Wang. Differential neural coding of acoustic flutter within primate auditory cortex. *Nature Neuroscience*, 10(6):763–771, April 2007.

[5] Marc Bolduc and Martin D. Levine. A review of biologically motivated space-variant data reduction models for robotic vision. *Comput. Vis. Image Underst.*, 69:170–184, February 1998.

[6] Alexandros Bouganis and Murray Shanahan. Training a spiking neural network to control a 4-dof robotic arm based on spike timing-dependent plasticity.

[7] R.M. Bradley. *Basic oral physiology*. Year Book Medical Publishers, 1981.

[8] Frederico Alan O. Cruz and Clia Martins Cortez. Computer simulation of a central pattern generator via kuramoto model. *Physica A: Statistical Mechanics and its Applications*, 353:258 – 270, 2005.

[9] Christina Enroth-Cugell and J. G. Robson. The contrast sensitivity of retinal ganglion cells of the cat. *The Journal of Physiology*, 187(3):517–552, 1966.

[10] A. K. Fidjeland and M. P. Shanahan. Accelerated simulation of spiking neural networks using gpus. In *Proc. IEEE International Joint Conference on Neural Networks*, July 2010.

[11] Paul Fitzpatrick. Yarp without yarp. `http://eris.liralab.it/yarpdoc/yarp_without_yarp.html`.

[12] V. Florian. Biologically inspired neural networks for the control of embodied agents.

[13] David Gamez, Richard Newcombe, Owen Holland, and Rob Knight. Two simulation tools for biologically inspired virtual robotics. In *Proceedings of the IEEE 5th Chapter Conference on Advances in Cybernetic Systems*, pages 85–90, 2006.

[14] J Gautrais and S Thorpe. Rate coding versus temporal order coding: a theoretical approach. *Bio Systems*, 48(1-3):57–65, 1998.

[15] Apostolos P. Georgopoulos, James Ashe, Nikolaos Smyrnis, and Masato Taira. The motor cortex and the coding of force. *Science*, 256(5064):1692–1695, 1992.

[16] Tim Gollisch, Markus Meister, Andreas Thiel, Jutta Kretzberg, Josef Ammermller, M. Greschner, C. W. Eurich, B. Werner, P. B. Cook, and C. L. Passaglia. Rapid neural coding in the retina with relative spike latencies. *Science*, 2010.

[17] Sten Grillner, Jeanette Hellgren, Ariane Mnard, Kazuya Saitoh, and Martin A. Wikstrm. Mechanisms for selection of basic motor programs - roles for the striatum and pallidum. *Trends in Neurosciences*, 28(7):364 – 370, 2005.

[18] Ellen C. Hildreth and John M. Hollerbach. *Artificial Intelligence: Computational Approach to Vision and Motor Control*. John Wiley and Sons, Inc., 2011.

[19] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve.

[20] Auke Jan Ijspeert, Alessandro Crespi, Dimitri Ryczko, and Jean-Marie Cabelguen. From swimming to walking with a salamander robot driven by a spinal cord model. *Science*, 315(5817):1416–1420, 2007.

[21] C. Koch. Biophysics of computation: Information processing in single neurons. *Journal of Computational Neuroscience*, 1999.

[22] Y. Kuramoto. Self-entrainment of a population of coupled non-linear oscillators. In H. Araki, editor, *Mathematical Problems in Theoretical Physics*, volume 39 of *Lecture Notes in Physics, Berlin Springer Verlag*, pages 420–422, 1975.

[23] R. Llinas. Neuron. *Scholarpedia*, 3(8):1490, 2008.

[24] Izhikevich Eugene M. Simple model of spiking neurons. *IEEE Transactions On Neural Networks*, 2003.

[25] Wolfgang Maass and Christopher M. Bishop, editors. *Pulsed Neural Networks*. MIT Press, 1999.

[26] Alexis Maldonado. `http://www.flickr.com/photos/amaldo/3754943096/`.

[27] Jaakko Malmivuo. *Bioelectromagnetism : principles and applications of bioelectric and biomagnetic fields*. Oxford University Press, New York, 1995.

[28] Giorgio Metta, Giulio Sandini, David Vernon, Lorenzo Natale, and Francesco Nori. The icub humanoid robot: an open platform for research in embodied cognition. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, PerMIS '08, pages 50–56, New York, NY, USA, 2008. ACM.

[29] Giorgio Metta, Giulio Sandini, David Vernon, Lorenzo Natale, and Francesco Nori. The icub humanoid robot: an open platform for research in embodied cognition. *PerMIS '08 Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, 2008.

[30] A. Novellino, P. D'Angelo, L. Cozzi, M. Chiappalone, V. Sanguineti, and S. Martinoia. Connecting neurons to a mobile robot: an in vitro bidirectional neural interface. *Intell. Neuroscience*, 2007:2–2, January 2007.

[31] F. Orabona, G. Metta, and G. Sandini. Object-based visual attention: a model for a behaving robot. In *Computer Vision and Pattern Recognition - Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on*, page 89, 2005.

[32] M. W. Oram and D. I. Perrett. Time course of neural responses discriminating different views of the face and head. *Journal of Neurophysiology*, 68(1):70–84, 1992.

[33] Dr Heather Read. Crash course on retinal synapses, pathways and processing. `http://read.uconn.edu/PSYC3501/Lecture04/`.

[34] Gennadiy Rozental. Boost test library: Unit test framework. `http://www.boost.org/doc/libs/1_35_0/libs/test/doc/components/utf/index.html`.

[35] Glenn S. Smith. Human color vision and the unsaturated blue color of the daytime sky. *American Journal of Physics*, 73(7):590–597, 2005.

[36] Dimitri van Heesch. Doxygen homepage. `http://www.stack.nl/~dimitri/doxygen/`.

[37] Rufin Van Rullen and Simon J. Thorpe. Rate coding versus temporal order coding: What the retinal ganglion cells tell the visual cortex. *Neural Comput.*, 13:1255–1283, June 2001.

[38] Stewart W. Wilson. On the retino-cortical mapping. *International Journal of Man-Machine Studies*, 18(4):361 – 389, 1983.

[39] Hiroyuki Yamamoto, Martin D. Levine, and Yehezkel Yeshurun. An active foveated vision system: attentional mechanisms and scan path convergence measures. *Comput. Vis. Image Underst.*, 63:50–65, January 1996.