**Project no.:** 027657

**Project full title:** Perception, Action & Cognition through Learning of Object-Action Complexes

**Project Acronym:** PACO-PLUS

# Deliverable no.: D6.1

# Title of the deliverable:

## *Implementation of RL and CL software modules for the PACO-PLUS system, debugged and tested and Actor-Critic closed loop modules for RL and CL*

| | |
|---|---|
| **Contractual Date of Delivery to the CEC:** | **31/01-07** |
| **Actual Date of Delivery to the CEC:** | **31/01-07** |
| **Organisation name of lead contractor for this deliverable:** | **BCCN** |
| **Author(s): BCCN** | |
| **Participants(s): BCCN** | |
| **Work package contributing to the deliverable:** | **WP6** |
| **Nature:** | **R/D** |
| **Version:** | **1.1** |
| **Total number of pages:** | **174** |
| Start date of project: | 1st Feb. 2006 **Duration:** 48 month |

| **Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)** | | |
|---|---|---|
| **Dissemination Level** | | |
| **PU** | Public | **X** |
| **PP** | Restricted to other programme participants (including the Commission Services) | |
| **RE** | Restricted to a group specified by the consortium (including the Commission Services) | |
| **CO** | Confidential, only for members of the consortium (including the Commission Services) | |

**Abstract:**

This deliverable describes various modules for implementing correlation-based (CL) as well as reinforcement learning (RL). It gives the full program code and is modular, such that users can implement these modules like any other C++ module. This documentation also exists on the public pages of the PACO home page.

Deliverable 6.1 contains the description and the C++ code for a program library that emulates various machine learning methods in a neuronal way (artificial neural networks). This library has been specifically developed and compiled for PACO+.

One goal of PACO+ is to draw conclusions from the existing biological models for cognitions (primates, humans) and learning ('all' animals with different degree of complexity) and employ the obtained results in an abstracted way on the robot's used in the project.

To this end it is required to investigate biological realistic learning methods in machines. For this step, it is not sufficient to just employ plain machine learning because these methods are too far away from the biological substrate and do not allow us "to learn from biology". (Having said this, it may, however, be that the final conclusion of this investigation step is that biological methods are not appropriate for the goals of cognitive robotics).

This led to the demand for a general purpose program library which emulates the most common biologically motivated neural network models in a modular way. All modules represent "neurons" with their attached learning methods taken from the literature and/or own work. The neurons can thus be connected in any desired ANN structure and will perform their algorithm in a robust (fully debugged) manner.

The deliverable contains a tutorial of how to use these artificial neurons.

Thus, it is now possible to implement different networks for the control of learning in different parts of the PACO+ agents without having to program the primary functions required.

# Neuronal Network Simulator Reference Manual

Jan 9 2007

# Contents

**1 A Neuronal Network simulator** **1**

1.1 Introduction . . . . . 1

1.2 Installation . . . . . 1

1.3 An example (OpenLoop.cpp) . . . . . 2

1.4 Compiling and linking . . . . . 5

1.5 Contact . . . . . 5

**2 Neuronal Network Simulator Hierarchical Index** **7**

2.1 Neuronal Network Simulator Class Hierarchy . . . . . 7

**3 Neuronal Network Simulator Class Index** **9**

3.1 Neuronal Network Simulator Class List . . . . . 9

**4 Neuronal Network Simulator File Index** **11**

4.1 Neuronal Network Simulator File List . . . . . 11

**5 Neuronal Network Simulator Class Documentation** **13**

5.1 Neuron Class Reference . . . . . 13

5.2 NeuronalNet Class Reference . . . . . 17

5.3 NeuronBP Class Reference . . . . . 21

5.4 NeuronEligibility Class Reference . . . . . 26

5.5 NeuronICO Class Reference . . . . . 29

5.6 NeuronISO Class Reference . . . . . 32

5.7 NeuronISO3 Class Reference . . . . . 34

5.8 NeuronNICO Class Reference . . . . . 37

5.9 NeuronPID Class Reference . . . . . 39

5.10 NeuronQ Class Reference . . . . . 44

5.11 NeuronRecruitment Class Reference . . . . . 48

5.12 NeuronSigmoid_exp Class Reference . . . . . 50

5.13 NeuronTD Class Reference . . . . . 53

5.14 RecDouble Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.15 Recordable Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.16 Recorder Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.17 Recruitment Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.18 Synapse Class Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.19 SynapseInput Class Reference . . . . . . . . . . . . . . . . . . . . . . . . 72
5.20 SynapseRecruitment Class Reference . . . . . . . . . . . . . . . . . . . . . 74

**6 Neuronal Network Simulator File Documentation 77**

6.1 Neuron.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.2 Neuron.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
6.3 NeuronalNet.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . 83
6.4 NeuronalNet.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . 87
6.5 NeuronBP.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . . 90
6.6 NeuronBP.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . 94
6.7 NeuronEligibility.cpp File Reference . . . . . . . . . . . . . . . . . . . . . 97
6.8 NeuronEligibility.h File Reference . . . . . . . . . . . . . . . . . . . . . . 99
6.9 NeuronICO.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . 101
6.10 NeuronICO.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . 103
6.11 NeuronISO.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . 105
6.12 NeuronISO.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . 107
6.13 NeuronISO3.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . 109
6.14 NeuronISO3.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . 111
6.15 NeuronNICO.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . 113
6.16 NeuronNICO.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . 115
6.17 NeuronPID.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . 117
6.18 NeuronPID.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . 120
6.19 NeuronQ.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . . 122
6.20 NeuronQ.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . 125
6.21 NeuronRecruitment.cpp File Reference . . . . . . . . . . . . . . . . . . . . 127
6.22 NeuronRecruitment.h File Reference . . . . . . . . . . . . . . . . . . . . . 128
6.23 NeuronSigmoid_exp.cpp File Reference . . . . . . . . . . . . . . . . . . . 130
6.24 NeuronSigmoid_exp.h File Reference . . . . . . . . . . . . . . . . . . . . 132
6.25 NeuronTD.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . 134
6.26 NeuronTD.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . . 136
6.27 OpenLoop.cpp File Reference . . . . . . . . . . . . . . . . . . . . . . . . 138
6.28 RecDouble.h File Reference . . . . . . . . . . . . . . . . . . . . . . . . . 140

6.29 Recordable.h File Reference . . . . . 142
6.30 Recorder.cpp File Reference . . . . . 144
6.31 Recorder.h File Reference . . . . . 147
6.32 Recruitment.h File Reference . . . . . 150
6.33 Synapse.cpp File Reference . . . . . 153
6.34 Synapse.h File Reference . . . . . 155
6.35 SynapseInput.cpp File Reference . . . . . 157
6.36 SynapseInput.h File Reference . . . . . 158
6.37 SynapseRecruitment.cpp File Reference . . . . . 160
6.38 SynapseRecruitment.h File Reference . . . . . 161

# Chapter 1

# A Neuronal Network simulator

## 1.1 Introduction

This simulator provides a huge variety of different correlations based and reinforcement learning rules. It aims to work as close as possible to a neuronal network with synapses and neurons as compartments. Thus you have to build up your neuronal network with neurons and synapse whereas a synapse can only detach from one neuron and attach only to another (or the identical) neuron. This scheme holds even for reinforcement learning, although the neuron itself acts like a maschine learning instance.

A neuron can consists of different attached synapses and the weights of these can change according to a learning rule. You can write your own rule or use a predefined one. A neuron can also transform the input for instance into a sigmoid function or a filter. Additionally a recruitment mechanism is build into the system. This gives you the possibility to simulate muscle- like behavior.

These networks can be used as Actor-Critic modules. Since each network has an input and an output chaining is possible, too.

This documentation is divided in two parts: how to install this simulator and how to use it (an example is given)

## 1.2 Installation

You have to set an environmental variable called **NNETPATH**. This can be done by means of three different approaches.

- If you will often use this simulator you maybe want to declare the variable in your ~/.bashrc (bash-shell) or ~/.cshrc (tc-shell):

  .bashrc:

  ```
  export NNETPATH=/home/kolo/NeuronalNet
  ```

  .cshrc:

  ```
  setenv NNETPATH /home/kolo/NeuronalNet
  ```

- If you don't want to change your ∗rc files you can run a script called **setVar**. You have to do this everytime you open a new shell.

```
/home/kolo/NeuronalNet/setVar
```

- Or you can write the directory in your Makefile or always include it when compiling

In all three cases replace the given directory with the directory of your Neuronal Network.

After you have set the variable you have to run a script called ./createLibrary.sh and wait some seconds.

```
/home/kolo/NeuronalNet/createLibrary.sh
```

This script serves as tool that you have to use each time you change the source code of the Neuronal Network.

## 1.3 An example (OpenLoop.cpp)

The example you will use is located in the directory **example** and is called OpenLoop.cpp. You can find these documentary you just read in raw format there. We will go through it step by step.

It is called OpenLoop due to the fact that it simulates a simple neuron with a differential Hebbian learning rule (called ICO-rule) in an open loop fashion. There is no feedback through the environment and you decide which inputs the neuron should get.

### 1.3.1 Initialization of the Neuronal Network

First you need an object of the class **NeuronalNet**. You will put all neurons and synapses into this file. It is important to set the type of all neurons to **Neuron** (and all synapses to **Synapse**) and only create them (with *new*) as a Neuron-type (or Synapse-type) you would like to use.

```
e.g. Neuron* nReflex = new NeuronBP(reflexF,reflexQ);
```

Since all neurons are of type Neuron you always have to cast them when using neuron-specific methods:

```
e.g. ((NeuronBP*)(nReflex))->setNormalize(reflexNorm);
```

When declaring synapses you have to set their descending neurons

```
e.g. Synapse* sReflex = new Synapse(nReflex);
```

and connect them to their destination neurons with the addSynapse(Synapse∗) method. One synapse can only recieve input from one neuron.

```
e.g. nICO->addSynapse(sReflex);
```

After creating and connecting your neurons you have to add them with the **addNeuron(neuron∗)** method or if they are input neurons with the **setInput(neuron∗)** method.

```
e.g. setInput(nReflex);
e.g. addNeuron(nICO);
```

Additionally you can set a neuron as output with the **setOutput(neuron∗)** method.

```
e.g. setOutput(nICO);
```

To communicate with your network you need two additional variables: an array with the same length as you have input neurons

```
e.g. double value[2];
```

and a pointer that will receive the output

```
e.g. double* output;
```

Now comes the example. First you need a network

```
NeuronalNet nnet;
```

Two neurons of the example are sensory neurons modeled as filter neurons:

```
Neuron* nReflex = new NeuronBP(reflexF,reflexQ);
((NeuronBP*)(nReflex))->setNormalize(reflexNorm);
nnet.setInput(nReflex);

Neuron* nPreflex = new NeuronBP(preflexF, preflexQ);
((NeuronBP*)(nPreflex))->setNormalize(preflexNorm);
nnet.setInput(nPreflex);
```

The third neuron you create is the learning neuron and is responsible for the plasticity of your synapses via the **calculate()** method. Each synapse has flags you can use in this method.

```
Neuron* nICO = new NeuronICO(learingRate);
```

Now as you have two input neurons and one output neuron you need two synapses. One of your synapses is the so called reflex input. It provides the neuron with an intrinsic behavior (even if everything is only open loop) and therefore the weight is initially 1 and the flag has to be 0.

```
Synapse* sReflex = new Synapse(nReflex);
sReflex->setWeight(sReflexWeight);
sReflex->setFlags(sReflexFlags);
```

The other synapse is the predictive one. At the beginning the weight should be 0 (this weight will be learned) and the flag is set to 1.

```
double sPreflexWeight = 0;
int    sPreflexFlags = 1;
Synapse* sPreflex = new Synapse(nPreflex);
sPreflex->setWeight(sPreflexWeight);
sPreflex->setFlags(sPreflexFlags);
```

The two synapses you have now created have to become connected to your ICO neuron via the **addSynapse(synapse∗)** method.

```
nICO->addSynapse(sReflex);
nICO->addSynapse(sPreflex);
```

At the end the learning neuron has to be added to the network (**addNeuron(neuron∗)**) and, since you want the easily see the output, assigned as a output neuron (**setOutput(neuron∗)**).

```
nnet.addNeuron(nICO);
nnet.setOutput(nICO);
```

### 1.3.2 Initialization of the Recorder

The simulator has a build-in recorder that you easily can use to write out all output values of your neurons and synapses, the weights of your synapses and all other arbitrary variables. Therefore you have to use the container **RecDouble**, because you can only add types that inherit from the type Recordable. This can be done with the **add(Recordable∗, string)** method. You can leave the string out, if you want. Finally you have to initalize the input (**init(string)**).

First you have to create a Recorder and assign a file name to it.

```
Recorder rec("data.dat");
```

The file will be overwritten if already exist and for each new file you need a new recorder.

Next you add all values out of your network you want to observe and add a small description to them.

```
rec.addValue(sPreflex->recWeight(), "Weight of Preflex");
rec.addValue(nPreflex, "Output of Preflex");
rec.addValue(((NeuronBP*)(nPreflex))->recOutput(), "Output of Synapse");
rec.addValue(nICO, "Output");
```

Additionally you record the plain input of the predictive input.

```
RecDouble preflex_input;
rec.addValue(&preflex_input, "Input to the preflex");
```

Last comes the initialization

```
rec.init("Weights of the preflex synapse during ICO\n# x0 is shut down at t = 6000");
```

### 1.3.3 Open loop simulation

Now, after you have prepared your network you can start to use it. You simply loop N-times and provide the network at each 200 time steps a peak pair with a 10 time steps inter-spike interval. The reflexive input should only last for 6000 time steps that we can investigate stability.

```
values[0] = (i<6000 ? (i%200==20 ? 1 : 0) : 0);
values[1] = (i%200==10 ? 1 : 0);
```

At each time step you have to fill your container with the proper value(s)

```
preflex_input.setValue(values[1]);
```

Now comes the place where the updating of your network and your recorder takes place. First you have to put your input into the network via the **setInputValues(double∗)** method. Next you update both via the **update()** method and last but not least you receive your output with the **getOutputValues()** method.

```
nnet.setInputValues(values);
nnet.update();
rec.update(i, N, N/10);
output = nnet.getOutputValues();
```

Now you can use this output for instance for a closed loop scenario or an Actor-Critic chaining.

One important line you should not forget in your own programm:

```
#include "nnet.h"
```

## 1.4 Compiling and linking

In the example directory you can find a Makefile which you can directly use to compile the example.

```
./make
```

If you want to use your own Makefile or compile it by hand you have to set the include directory

```
-I/home/kolo/NeuronalNet/include
```

and the libary path

```
-lm /home/kolo/NeuronalNet/lib/nnet.a
```

as an option.

Have fun!

## 1.5 Contact

The concept was build by Daniel Steingrube (daniel_at_bccn-goettingen.de), Steffen Wischmann (steffen_-at_bccn-goettingen.de) and Christoph Kolodziejski (kolo_at_bccn-goettingen.de). The C++ version was mainly created by Christoph Kolodziejski and the Java version by Daniel Steingrube. Here, only the C++ version is provided but the Java version is available, too. Ask Daniel Steingrube or Christoph Kolodziejski for the source code.

The source code can be found on the PACO PLUS website: http://www.paco-plus.org/Public+Deliverables

# Chapter 2

# Neuronal Network Simulator Hierarchical Index

## 2.1 Neuronal Network Simulator Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:


- NeuronalNet . . . . . 17
- Recordable . . . . . 59
  - Neuron . . . . . 13
    - NeuronICO . . . . . 29
      - NeuronISO . . . . . 32
      - NeuronISO3 . . . . . 34
      - NeuronNICO . . . . . 37
    - NeuronQ . . . . . 44
    - NeuronRecruitment . . . . . 48
      - NeuronBP . . . . . 21
      - NeuronEligibility . . . . . 26
      - NeuronPID . . . . . 39
      - NeuronSigmoid_exp . . . . . 50
    - NeuronTD . . . . . 53
  - RecDouble . . . . . 57
  - Synapse . . . . . 68
    - SynapseInput . . . . . 72
    - SynapseRecruitment . . . . . 74
- Recorder . . . . . 60
- Recruitment . . . . . 64
  - NeuronRecruitment . . . . . 48

# Chapter 3

# Neuronal Network Simulator Class Index

## 3.1 Neuronal Network Simulator Class List

Here are the classes, structs, unions and interfaces with brief descriptions:


Neuron (Basic Neuron ) . . . . . . . . . . 13
NeuronalNet (Basic Neuronal Network that contains all compartments ) . . . . . . . . . . 17
NeuronBP (Bandpass Neuron ) . . . . . . . . . . 21
NeuronEligibility (Eligibility Neuron ) . . . . . . . . . . 26
NeuronICO (ICO Neuron ) . . . . . . . . . . 29
NeuronISO (ISO Neuron ) . . . . . . . . . . 32
NeuronISO3 (ISO-3 Neuron ) . . . . . . . . . . 34
NeuronNICO (Normalized ICO Neuron ) . . . . . . . . . . 37
NeuronPID (PID Neuron ) . . . . . . . . . . 39
NeuronQ (Q-Learning Neuron ) . . . . . . . . . . 44
NeuronRecruitment (Recruitment Neuron ) . . . . . . . . . . 48
NeuronSigmoid_exp (Sigmoid Neuron (exp) ) . . . . . . . . . . 50
NeuronTD (TD-Learning Neuron ) . . . . . . . . . . 53
RecDouble (Double-valued Recordable ) . . . . . . . . . . 57
Recordable (Interface for Recording ) . . . . . . . . . . 59
Recorder (Simple Recorder ) . . . . . . . . . . 60
Recruitment (Base Recruitment ) . . . . . . . . . . 64
Synapse (Basic Synapse ) . . . . . . . . . . 68
SynapseInput (Sensory (Input) Synapse ) . . . . . . . . . . 72
SynapseRecruitment (Recruitment Synapse ) . . . . . . . . . . 74

# Chapter 4

# Neuronal Network Simulator File Index

## 4.1 Neuronal Network Simulator File List

Here is a list of all files with brief descriptions:


Neuron.cpp . . . 77
Neuron.h . . . 80
NeuronalNet.cpp . . . 83
NeuronalNet.h . . . 87
NeuronBP.cpp . . . 90
NeuronBP.h . . . 94
NeuronEligibility.cpp . . . 97
NeuronEligibility.h . . . 99
NeuronICO.cpp . . . 101
NeuronICO.h . . . 103
NeuronISO.cpp . . . 105
NeuronISO.h . . . 107
NeuronISO3.cpp . . . 109
NeuronISO3.h . . . 111
NeuronNICO.cpp . . . 113
NeuronNICO.h . . . 115
NeuronPID.cpp . . . 117
NeuronPID.h . . . 120
NeuronQ.cpp . . . 122
NeuronQ.h . . . 125
NeuronRecruitment.cpp . . . 127
NeuronRecruitment.h . . . 128
NeuronSigmoid_exp.cpp . . . 130
NeuronSigmoid_exp.h . . . 132
NeuronTD.cpp . . . 134
NeuronTD.h . . . 136
OpenLoop.cpp . . . 138
RecDouble.h . . . 140
Recordable.h . . . 142
Recorder.cpp . . . 144
Recorder.h . . . 147
Recruitment.h . . . 150
Synapse.cpp . . . 153

Synapse.h . . . . . . . . . . . . 155
SynapseInput.cpp . . . . . . . . . . . . 157
SynapseInput.h . . . . . . . . . . . . 158
SynapseRecruitment.cpp . . . . . . . . . . . . 160
SynapseRecruitment.h . . . . . . . . . . . . 161

# Chapter 5

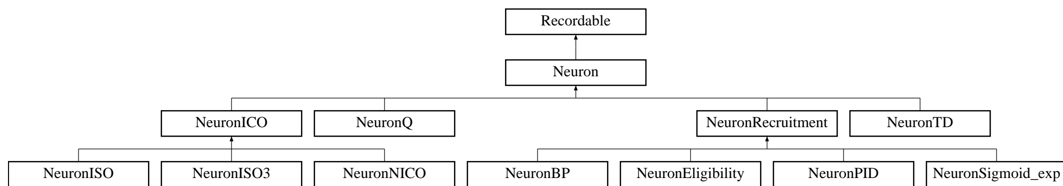
# Neuronal Network Simulator Class Documentation

## 5.1 Neuron Class Reference

Basic Neuron.

```
#include <Neuron.h>
```

Inheritance diagram for Neuron::




### Public Member Functions

- Neuron ()
- virtual ∼Neuron ()
- void addSynapse (Synapse ∗synapse)

    *Add a synapse to this neuron.*

- virtual void reset ()

    *Reset the neuron to zero.*

- virtual void calculate ()

    *Calculation of the next output.*

- virtual void update ()

    *Updating after calculation.*

- void clear ()

*Destruction of the neuron.*

- void setOutput (double output)
- double getOutput ()
- double record ()

## Protected Attributes

- std::list< Synapse * > synapsis_
- double output_
- double nextoutput_

### 5.1.1 Detailed Description

This is the base neuron all other neurons inherit from. Basically it sums up the weighted values of all connected synapses.

First this summation is calculated for the whole network followed by an updating step.

**Date:**
12/20/2006 02:02:44 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 Neuron::Neuron ()

Constructor

```
35 {
36 }
```

#### 5.1.2.2 Neuron::~Neuron () [virtual]

Destructor

```
40 {
41   clear();
42 }
```

### 5.1.3 Member Function Documentation

#### 5.1.3.1 void Neuron::addSynapse (Synapse * *synapse*)

Add a synapse to this neuron

**Parameters:**
*synapse* Synapse* - A pointer to a synapse that will be connected

```
60 {
61   synapsis_.push_back(synapse);
62 }
```

#### 5.1.3.2 void Neuron::reset () [virtual]

Variable **output_** and **nextoutput_** are set 0

Reimplemented in NeuronBP, NeuronPID, and NeuronQ.

```
75 {
76   output_ = 0;
77   nextoutput_ = 0;
78 }
```

#### 5.1.3.3 void Neuron::calculate () [virtual]

Iterating the connected synapses and summing up their output weighted with their weight The output is stored into the variable **nextoutput_**

**Remarks:**
Use always the variable **nextoutput_**

Reimplemented in NeuronBP, NeuronEligibility, NeuronICO, NeuronISO, NeuronISO3, NeuronNICO, NeuronPID, NeuronQ, NeuronSigmoid_exp, and NeuronTD.

```
92 {
93   nextoutput_ = 0;
94   std::list<Synapse*>::iterator i, iend = synapsis_.end();
95   for(i = synapsis_.begin(); i!=iend; ++i)
96   {
97     nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
98   }
99 }
```

#### 5.1.3.4 void Neuron::update () [virtual]

After all neurons within the Neuronal Network are calculated they become updated **nextoutput_** -> **output_**

Reimplemented in NeuronBP.

```
112 {
113   output_ = nextoutput_;
114 }
```

#### 5.1.3.5 void Neuron::clear ()

Destruction of all connected synapses and the neuron itself

**Remarks:**
The complete neural network destruct itself recursive thus you don't have to do this by hand

```
129 {
130   nextoutput_ = 0;
131   std::list<Synapse*>::iterator i, imax = synapsis_.end();
132   for(i = synapsis_.begin(); i != imax; ++i)
133     delete (*i);
134   synapsis_.clear();
135 }
```

#### 5.1.3.6 void Neuron::setOutput (double *output*) [inline]

```
111 {output_ = output;};
```

#### 5.1.3.7 double Neuron::getOutput () [inline]

```
112 {return output_;};
```

#### 5.1.3.8 double Neuron::record () [inline, virtual]

Implements Recordable.

```
114 {return output_;}
```

### 5.1.4 Member Data Documentation

#### 5.1.4.1 std::list<Synapse*> Neuron::synapsis_ [protected]

A list of synapses

#### 5.1.4.2 double Neuron::output_ [protected]

The output of the neuron

#### 5.1.4.3 double Neuron::nextoutput_ [protected]

The next output of the neuron that is determined during the calculating step and set to output during the update step

The documentation for this class was generated from the following files:

- Neuron.h
- Neuron.cpp

## 5.2 NeuronalNet Class Reference

Basic Neuronal Network that contains all compartments.

```
#include <NeuronalNet.h>
```

### Public Member Functions

- NeuronalNet ()
- ~NeuronalNet ()
- void addNeuron (Neuron ∗neuron)

    *Add a neuron to the Neuronal Network.*

- void setOutput (Neuron ∗neuron)

    *Add a neuron as an output to the Neuronal Network.*

- void setInput (Neuron ∗neuron)

    *Add a neuron as an input to the Neuronal Network.*

- void setInputValues (double values[ ])

    *Set the input of the Neuronal Network.*

- double ∗ getOutputValues ()

    *Get the output of the Neuronal Network.*

- void update ()

    *Compute and update all neurons.*

- void reset ()

    *Reset the Neuronal Network.*

- void clear ()

    *Destruction of the Neuronal Network.*

### Private Attributes

- std::list< Neuron ∗ > neurons_
- std::list< SynapseInput ∗ > inputs_
- std::list< Neuron ∗ > outputs_

### 5.2.1 Detailed Description

All compartments (Neurons and Synapses) are put into the Neuronal Network.

During the calculation the next output of each neuron is computed.

Afterwards during the update phase all calculated values are set to the output of the neurons.

**Date:**
01/03/2007 03:49:12 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.2.2 Constructor & Destructor Documentation

#### 5.2.2.1 NeuronalNet::NeuronalNet ()

Constructor

```
38 {
39 }
```

#### 5.2.2.2 NeuronalNet::∼NeuronalNet ()

Destructor

```
42 {
43   clear();
44 }
```

### 5.2.3 Member Function Documentation

#### 5.2.3.1 void NeuronalNet::addNeuron (Neuron ∗ *neuron*)

Add a neuron to the Neuronal Network

**Parameters:**
 ***neuron*** Neuron∗ - A pointer to a neuron that will be calculated

```
103 {
104   neurons_.push_back(neuron);
105 }
```

#### 5.2.3.2 void NeuronalNet::setOutput (Neuron ∗ *neuron*)

Add a neuron as an output to the Neuronal Network

**Parameters:**
 ***neuron*** Neuron∗ - A pointer to a neuron that will be calculated

**Remarks:**
 The output can be easily used outside
 A neuron should additionally added as an output.

**See also:**
 addNeuron

```
143 {
144   outputs_.push_back(neuron);
145 }
```

#### 5.2.3.3 void NeuronalNet::setInput (Neuron ∗ *neuron*)

Add a neuron as an input to the Neuronal Network

**Parameters:**
 ***neuron*** Neuron∗ - A pointer to a neuron that will be calculated

**Remarks:**
 The input to this neuron is set from outside.
 A neuron should only be added either as input or as an usual neuron.

**See also:**
addNeuron

```
122 {
123   addNeuron(neuron);
124   Synapse* s = new SynapseInput();
125   inputs_.push_back((SynapseInput*)(s));
126   neuron->addSynapse(s);
127 }
```

#### 5.2.3.4 void NeuronalNet::setInputValues (double *values*[ ])

Set the input values of all input neurons that are within the Neuronal Network.

**Parameters:**
***values[ ]*** double - An array that has the same size as input neurons embedded

**Remarks:**
There is no size check! You have to take by yourself.

```
159 {
160   double* cur = values;
161   std::list<SynapseInput*>::iterator i, iend = inputs_.end();
162   for(i = inputs_.begin(); i!=iend; ++i)
163   {
164     (*i)->setOutput(*cur);
165     cur++;
166   }
167
168 }
```

#### 5.2.3.5 double * NeuronalNet::getOutputValues ()

Get the output values of all neurons that were assigned as output neurons.

**Returns:**
double* - A pointer with all output values stored

**Remarks:**
There is no information about the size of the array! You have to rememeber by yourself

```
181 {
182   double* out = new double[outputs_.size()];
183   double* cur = out;
184
185   std::list<Neuron*>::iterator i, iend = outputs_.end();
186   for(i = outputs_.begin(); i!=iend; ++i)
187   {
188     *cur = (*i)->getOutput();
189     cur++;
190   }
191
192   return out;
193 }
```

#### 5.2.3.6 void NeuronalNet::update ()

Iterate all neurons. First call their **calculate()** method and after all the output values were computed call the **update()** method.

**Remarks:**
Mostly the **update()** method only sets the value of the output to the calculated value.

```
80 {
81   std::list<Neuron*>::iterator i, iend = neurons_.end();
82   for(i = neurons_.begin(); i!=iend; ++i)
83   {
84     (*i)->calculate();
85   }
86   for(i = neurons_.begin(); i!=iend; ++i)
87   {
88     (*i)->update();
89   }
90 }
```

#### 5.2.3.7 void NeuronalNet::reset ()

Iterate all neurons and call their reset function.

```
60 {
61   std::list<Neuron*>::iterator i, iend = neurons_.end();
62   for(i = neurons_.begin(); i!=iend; ++i)
63   {
64     (*i)->reset();
65   }
66 }
```

#### 5.2.3.8 void NeuronalNet::clear ()

Iterate all neurons and call the **clear()** method.

**Remarks:**
The complete neural network destruct itself recursive thus you don't have to do this by hand.

```
205 {
206
207   std::list<Neuron*>::iterator i, imax = neurons_.end();
208   for(i = neurons_.begin(); i != imax; ++i)
209     delete (*i);
210   neurons_.clear();
211   inputs_.clear();
212 }
```

### 5.2.4 Member Data Documentation

#### 5.2.4.1 std::list<Neuron*> NeuronalNet::neurons_ [private]

A list of all neurons except the input neurons

#### 5.2.4.2 std::list<SynapseInput*> NeuronalNet::inputs_ [private]

A list of all input neurons

#### 5.2.4.3 std::list<Neuron*> NeuronalNet::outputs_ [private]

A list of all neurons that are outputs

The documentation for this class was generated from the following files:

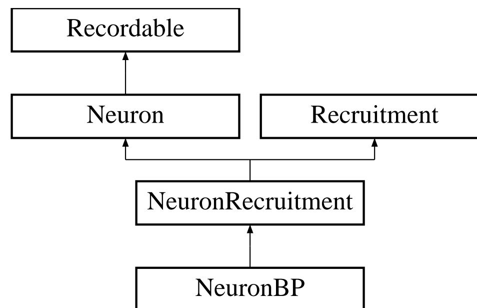
- NeuronalNet.h
- NeuronalNet.cpp

## 5.3 NeuronBP Class Reference

Bandpass Neuron.

`#include <NeuronBP.h>`

Inheritance diagram for NeuronBP::




### Public Member Functions

- NeuronBP (double f=DEF_F, double q=DEF_Q)

  *Constructor that sets frequency and quality.*

- void calculate ()

  *Calculation of the next output.*

- void calculate (double in)

  *Calculation of the next value.*

- void update ()

  *Updating of the neuron.*

- void reset ()

  *Reseting of the neuron.*

- void setFQ (double f, double q)

  *Initialization.*

- void calcNorm ()

  *Calculation of the normalizing factor.*

- void setNormalize (bool fnormalize)
- bool getNormalize ()

### Private Attributes

- double denominator_ [2]
- double buffer_ [2]
- double norm_
- bool normalize_

### Static Private Attributes

- static const double DEF_F = 0.1
- static const double DEF_Q = 0.51

### 5.3.1 Detailed Description

A bandpass neuron can emulate a filter according to:

$$h(t) = \frac{1}{b} e^{at} \sin(bt)$$

with $a = \frac{\pi f}{q}$ and $b = \sqrt{(2\pi f)^2 - a^2}$ yields into:

$$y[(n+1)T] = -2 \cdot e^{-aT} cosbTy[nY] - e^{-2 \cdot aT} y[(n-1)T] + \frac{1}{b} e^{aT} \sin(bT) x[nT]$$

The last factor $x[nT]$ is set to 1 (irrelevant because of the normalization) and T is set to 1, too.

**f** (default) = 0.1

**q** (default) = 0.51

**normlize** (default) = true

The higher the frequency the sharper the peak

The higher the quality the more is the filter oscillating

**Date:**
01/03/2007 09:26:11 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 NeuronBP::NeuronBP (double *f* = DEF_F, double *q* = DEF_Q)

Constructor

```
66 {
67   reset();
68   normalize_ = true;
69   setFQ(f,q);
70 }
```

### 5.3.3 Member Function Documentation

#### 5.3.3.1 void NeuronBP::calculate () [virtual]

Calculation of the **nextoutput_** either without recruitment or with recruitment: no recruitment: The value provided by the connected synapses are put into the filter as a new value recruitment: The value provided by the connected synapses are put into the recruitment mechanism (**fibre**) and only the entity is put into the filter as a new value.

Reimplemented from Neuron.

```
204 {
205   Neuron::calculate();
206
207   if(getIsRecruitment() && nextoutput_)
208   {
209     setFibres(nextoutput_);
210     calculate(getEntity());
211   }
212   else
213   {
214     calculate(nextoutput_);
215   }
216
217 }
```

#### 5.3.3.2 void NeuronBP::calculate (double *in*)

Calculation of the next value according to the filter equations.

**Parameters:**

*in* double - The new value.

```
170 {
171
172   nextoutput_ = in
173               - denominator_[0] * buffer_[0]
174               - denominator_[1] * buffer_[1];
175
176   buffer_[1] = buffer_[0];
177   buffer_[0] = nextoutput_;
178
179
180   if (normalize_)
181   {
182     nextoutput_ /= norm_;
183   }
184
185
186 }
```

#### 5.3.3.3 void NeuronBP::update () [virtual]

Updating of the neuron either without recruitment (nothing changed compared to Neuron) or with recruitment: The calculated value is put as the normalized one into the learning pathway (**normalized output**) and the output weighted with the fibres is set as 'real' output.

Because of the feedback the number of used fibres may have changed.

Reimplemented from Neuron.

```
235 {
236   Neuron::update();
237
238   setNormalizedoutput(output_);
239   if(getIsRecruitment())
240   {
241     checkFeedback();
242     output_ *= getFibres();
243   }
244
245 }
```

#### 5.3.3.4 void NeuronBP::reset () [virtual]

Reseting of the neuron.

Reimplemented from Neuron.

```
256 {
257   buffer_[0] = 0;
258   buffer_[1] = 0;
259   Neuron::reset();
260   output_ = 0;
261   nextoutput_ = 0;
262 }
```

#### 5.3.3.5 void NeuronBP::setFQ (double *f*, double *q*)

Initialization of the **denomiator_** needed for the calculation of the filter.

**Parameters:**

*f* double - The frequency

*q* double - The quality

**Remarks:**

The quality should be bigger than 0.51

```
88 {
89   // If Q is ok
90   if (q > 0)
91   {
92
93     double fTimesPi = f * M_PI * 2.;
94     double e = fTimesPi / (2. * q);
95
96     // If root is ok
97     if ((fTimesPi * fTimesPi - e * e) > 0)
98     {
99
100       double w = sqrt(fTimesPi * fTimesPi - e * e);
101       denominator_[0] = -2. * exp(-e) * cos(w);
102       denominator_[1] = exp(-2. * e);
103
104       calcNorm();
105
106     }
107     // If root is bac
108     else
109     {
110       std::cerr << "ERROR - bandpass: The quality is too low" << std::endl;
111       exit(1);
112     }
113
114   }
115   // If Q is bad
116   else
117   {
118     std::cerr << "bandpass: The quality is rubbish!" << std::endl;
119     exit(1);
120   }
121 }
```

#### 5.3.3.6 void NeuronBP::calcNorm ()

Calculation of the normalizing factor.

**Remarks:**

The 'search' for the maximum is limited to 200. This can cause trouble if the frequency is to low.

```
133 {
134   norm_ = 1.;
135   // Reset of buffer
136   for (int i = 0; i < 2; i++)
137   {
138     buffer_[i] = 0;
139   }
140   // Calculation of new norm
141   double nnorm = 0;
142   for (int i = 0; i < 200; i++)
143   {
```

```
144     calculate((i == 5) ? 1 : 0);
145     nnorm = max(nnorm, nextoutput_);
146   }
147   nextoutput_ = 0;
148
149   if (nnorm != 0)
150     norm_ = nnorm;
151
152   // Reset of buffer
153   for (int i = 0; i < 2; i++)
154   {
155     buffer_[i] = 0;
156   }
157 }
```

#### 5.3.3.7 void NeuronBP::setNormalize (bool *fnormalize*) [inline]

```
145 {normalize_ = fnormalize;};
```

#### 5.3.3.8 bool NeuronBP::getNormalize () [inline]

```
146 {return normalize_;};
```

### 5.3.4 Member Data Documentation

#### 5.3.4.1 double NeuronBP::denominator_[2] [private]

The pre-factor

#### 5.3.4.2 double NeuronBP::buffer_[2] [private]

The output with history

#### 5.3.4.3 double NeuronBP::norm_ [private]

The normalizing factor

#### 5.3.4.4 bool NeuronBP::normalize_ [private]

A switch for normalizing

#### 5.3.4.5 const double NeuronBP::DEF_F = 0.1 [static, private]

#### 5.3.4.6 const double NeuronBP::DEF_Q = 0.51 [static, private]

The documentation for this class was generated from the following files:

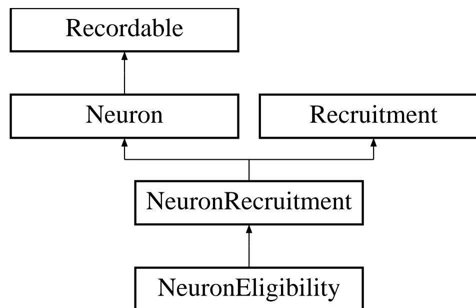
- NeuronBP.h
- NeuronBP.cpp

## 5.4 NeuronEligibility Class Reference

Eligibility Neuron.

`#include <NeuronEligibility.h>`

Inheritance diagram for NeuronEligibility::




### Public Member Functions

- NeuronEligibility ()
- NeuronEligibility (double eligibility)
- void calculate ()

  *Calculation of the next output.*

### Private Attributes

- double eligibility_
- double lastoutput_

### 5.4.1 Detailed Description

An eligibility neuron emulates an eligibility trace used for Reinforcement learning techniques.

If the eligibility value is set to zero only the next 4 steps are unequal 0. Otherwise it decays exponential with the eligibility value. **eligibility value** (default) = 0

**Date:**
01/07/2007 05:40:17 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.4.2 Constructor & Destructor Documentation

#### 5.4.2.1 NeuronEligibility::NeuronEligibility () `[inline]`

Constructor

```
52 :eligibility_(0.),lastoutput_(0.){};
```

#### 5.4.2.2 NeuronEligibility::NeuronEligibility (double *eligibility*) [inline]

```
54 :eligibility_(eligibility),lastoutput_(0){};
```

### 5.4.3 Member Function Documentation

#### 5.4.3.1 void NeuronEligibility::calculate () [virtual]

Calculation of the **nextoutput_** using a eligibility method.

**Remarks:**

If the value of eligibility is 0 the neuron stores a value only for two step further. Otherwise the buffered value decays until it reach 0.1.

Reimplemented from Neuron.

```
48 {
49
50   Neuron::calculate();
51
52   nextoutput_ += lastoutput_;
53
54   // if there is a value unequal 0
55   if(eligibility_)
56   {
57
58     lastoutput_ = eligibility_ * nextoutput_;
59     if(lastoutput_ < 0.01)
60     {
61       lastoutput_ = 0;
62     }
63
64   }
65   // if the value is equal 0
66   else
67   {
68     if(lastoutput_ == 0.5)
69     {
70       lastoutput_ = 0.25;
71     }
72     else if(lastoutput_ == 0.25)
73     {
74       lastoutput_ = 0.1;
75     }
76     else if(lastoutput_ == 0.1)
77     {
78       lastoutput_ = 0;
79     }
80     else
81     {
82       lastoutput_ = 0.5 * nextoutput_;
83     }
84   }
85
86
87 }
```

### 5.4.4 Member Data Documentation

#### 5.4.4.1 double NeuronEligibility::eligibility_ [private]

The eligiblity factor

#### 5.4.4.2 double NeuronEligibility::lastoutput_ [private]

The buffer of the neuron

The documentation for this class was generated from the following files:

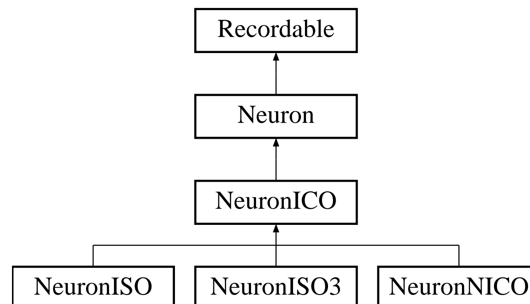
- NeuronEligibility.h
- NeuronEligibility.cpp

## 5.5 NeuronICO Class Reference

ICO Neuron.

```
#include <NeuronICO.h>
```

Inheritance diagram for NeuronICO::




### Public Member Functions

- NeuronICO ()
- NeuronICO (double learningRate)

    *Constructor that sets the learning rate.*

- virtual void calculate ()

    *Calculation of the next output.*

- void setLearningRate (double learningRate)
- double getLearningRate ()
- void setNoLearning (bool noLearning)
- bool getNoLearning ()

### Protected Attributes

- double learningRate_
- double reflex_
- bool noLearning_

### 5.5.1 Detailed Description

A ICO Neuron that inherit from Neuron.

Learning rule:

$$\Delta\rho_i(t) = \mu \cdot u_i(t) \cdot u_0^{'}(t)$$

where $\mu$ is the learning rate

and $\rho_i$ is the weight

and $u_i$ with $i = 1..n$ are the predictive inputs

and $u_0$ is the reflex input.

**learning** rate (default) = 1e-4

**Date:**
01/03/2007 08:07:27 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.5.2 Constructor & Destructor Documentation

### 5.5.2.1 NeuronICO::NeuronICO ()

Constructor

```
44 {
45
46   reset();
47   learningRate_ = 1e-4;
48   noLearning_ = 0;
49   reflex_ = 0;
50 }
```

### 5.5.2.2 NeuronICO::NeuronICO (double *learningRate*)

Constructor that sets the learning rate.

```
58 {
59   reset();
60   learningRate_ = learningRate;
61   noLearning_ = 0;
62   reflex_ = 0;
63 }
```

## 5.5.3 Member Function Documentation

### 5.5.3.1 void NeuronICO::calculate () `[virtual]`

Iterates all connected synapses and calculates the **nextoutput_** according to the ICO rule.

**Remarks:**

The flags are:
**0** <=> Reflex input (will not be learned)
**1** <=> Predictive input (will be learned)

Reimplemented from Neuron.

Reimplemented in NeuronISO, NeuronISO3, and NeuronNICO.

```
81 {
82
83   // Iterate all synapses and sum them up into nextoutput_
84   nextoutput_ = 0;
85   double nextreflex = 0;
86   std::list<Synapse*>::iterator i, iend = synapsis_.end();
87   for(i = synapsis_.begin(); i!=iend; ++i)
88   {
89     nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
90     if (((*i)->getFlags()) == 0)
91     {
92       nextreflex += (*i)->getOutput();
93     }
94   }
95
96   if(noLearning_)
97     return;
98
99   // Learn
100   double derivReflex = nextreflex - reflex_;
101
```

```
102   for(i = synapsis_.begin(); i!=iend; ++i)
103   {
104     if (((*i)->getFlags()) == 1)
105     {
106 
107       (*i)->setWeight((*i)->getWeight() + learningRate_ * derivReflex
108             * (*i)->getFactor() * (*i)->getOutput());
109 
110     }
111   }
112   reflex_ = nextreflex;
113 }
```

#### 5.5.3.2 void NeuronICO::setLearningRate (double *learningRate*) [inline]

```
99 {learningRate_ = learningRate;};
```

#### 5.5.3.3 double NeuronICO::getLearningRate () [inline]

```
100 {return learningRate_;};
```

#### 5.5.3.4 void NeuronICO::setNoLearning (bool *noLearning*) [inline]

```
102 {noLearning_ = noLearning;};
```

#### 5.5.3.5 bool NeuronICO::getNoLearning () [inline]

```
103 {return noLearning_;};
```

### 5.5.4 Member Data Documentation

#### 5.5.4.1 double NeuronICO::learningRate_ [protected]

The learning rate

#### 5.5.4.2 double NeuronICO::reflex_ [protected]

The reflex input

#### 5.5.4.3 bool NeuronICO::noLearning_ [protected]

A switch for (no) learning

The documentation for this class was generated from the following files:

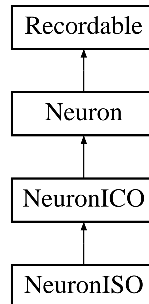
- NeuronICO.h
- NeuronICO.cpp

## 5.6 NeuronISO Class Reference

ISO Neuron.

```
#include <NeuronISO.h>
```

Inheritance diagram for NeuronISO::




### Public Member Functions

- NeuronISO ()
- NeuronISO (double learningRate)
- void calculate ()

  *Calculation of the next output.*

### Private Attributes

- double preflex_

### 5.6.1 Detailed Description

A ISO Neuron that inherit from Neuron or ICO Neuron, respectively.

Learning rule:

$$\Delta\rho_i(t) = \mu \cdot u_i(t) \cdot v^{'}(t)$$

where $\mu$ is the learning rate

and $\rho_i$ is the weight

and $u_i$ with $i = 1..n$ are the predictive inputs

and $v$ is the output.

**See also:**
Neuron
NeuronICO

**Date:**
01/03/2007 07:37:33 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.6.2 Constructor & Destructor Documentation

### 5.6.2.1 NeuronISO::NeuronISO () [inline]

Constructor

```
60 {};
```

### 5.6.2.2 NeuronISO::NeuronISO (double *learningRate*) [inline]

```
62 :NeuronICO(learningRate){};
```

## 5.6.3 Member Function Documentation

### 5.6.3.1 void NeuronISO::calculate () [virtual]

Iterates all connected synapses and calculates the **nextoutput_** according to the ISO rule.

**Remarks:**

The flags are:
**0** <=> Reflex input (will not be learned)
**1** <=> Predictive input (will be learned)

Reimplemented from NeuronICO.

```
55 {
56
57   // Iterate all synapses and sum them up into nextoutput_
58   nextoutput_ = 0;
59   std::list<Synapse*>::iterator i, iend = synapsis_.end();
60   for(i = synapsis_.begin(); i!=iend; ++i)
61   {
62     nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
63   }
64
65   if(noLearning_)
66     return;
67
68   // Learning
69   double derivOutput = nextoutput_ - output_;
70   for(i = synapsis_.begin(); i!=iend; ++i)
71   {
72     if ((*i)->getFlags() == 1)
73     {
74       (*i)->setWeight((*i)->getWeight() + learningRate_ * derivOutput
75             * (*i)->getOutput());
76
77     }
78   }
79 }
```

## 5.6.4 Member Data Documentation

### 5.6.4.1 double NeuronISO::preflex_ [private]

The predictive input

The documentation for this class was generated from the following files:

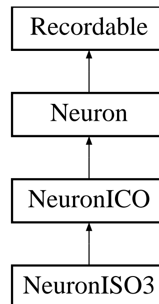
- NeuronISO.h
- NeuronISO.cpp

# 5.7 NeuronISO3 Class Reference

ISO-3 Neuron.

```
#include <NeuronISO3.h>
```

Inheritance diagram for NeuronISO3::




## Public Member Functions

- NeuronISO3 ()
- NeuronISO3 (double learningRate)
- void calculate ()

  *Calculation of the next output.*

## Private Attributes

- double preflex_

### 5.7.1 Detailed Description

A ISO-3 Neuron that inherit from Neuron or ICO Neuron, respectively.

Learning rule:

$$\Delta\rho_i(t) = \mu \cdot r(t) \cdot u_i(t) \cdot v^{'}(t)$$

where $\mu$ is the learning rate

and $\rho_i$ is the weight

and $u_i$ with $i = 1..n$ are the predictive inputs

and $v$ is the output

and $r$ is the relevance input.

**learning** rate (default) = 1e-4

**Date:**
01/03/2007 08:46:05 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.7.2 Constructor & Destructor Documentation

#### 5.7.2.1 NeuronISO3::NeuronISO3 () `[inline]`

Constructor

```
79 {};
```

#### 5.7.2.2 NeuronISO3::NeuronISO3 (double *learningRate*) [inline]

```
81 :NeuronICO(learningRate){};
```

## 5.7.3 Member Function Documentation

### 5.7.3.1 void NeuronISO3::calculate () [virtual]

Iterates all connected synapses and calculates the **nextoutput_** according to the ISO3 rule.

**Returns:**

-

**Remarks:**

The flags are:
**0** <=> Reflex input (will not be learned)
**1** <=> Predictive input (will be learned)
**2** <=> Driving input (will not be learned)
**3** <=> Relevance input (will not be learned and not transmitted)

Reimplemented from NeuronICO.

```
59 {
60
61   // Iterate all synapses and sum them up into nextoutput_
62   nextoutput_ = 0;
63   std::list<Synapse*>::iterator i, iend = synapsis_.end();
64   double R=1;
65   double D=1;
66   for(i = synapsis_.begin(); i!=iend; ++i)
67   {
68     if ((*i)->getFlags() != 3)
69     {
70       nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
71     }
72
73     if ((*i)->getFlags() == 2)
74     {
75       D *= (*i)->getWeight() * (*i)->getOutput();
76     }
77
78     if ((*i)->getFlags() == 3)
79     {
80       R *= (*i)->getWeight() * (*i)->getOutput();
81       if(R < 0)
82         R = 0;
83
84     }
85
86   }
87
88   // No negative current is allowed
89   if(nextoutput_ < 0)
90   {
91     nextoutput_ = 0;
92   }
93
94   if(noLearning_)
95     return;
96
97   // Learn
```

```
98   double derivOutput = nextoutput_ - output_;
99   for(i = synapsis_.begin(); i!=iend; ++i)
100    {
101      if ((*i)->getFlags() == 1)
102      {
103        (*i)->setWeight((*i)->getWeight() + learningRate_ * derivOutput
104              * (*i)->getOutput() * R * D);
105
106      }
107    }
108  }
```

### 5.7.4 Member Data Documentation

#### 5.7.4.1 double NeuronISO3::preflex_ [private]

The documentation for this class was generated from the following files:

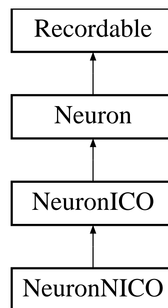
- NeuronISO3.h
- NeuronISO3.cpp

## 5.8 NeuronNICO Class Reference

Normalized ICO Neuron.

`#include <NeuronNICO.h>`

Inheritance diagram for NeuronNICO::




### Public Member Functions

- NeuronNICO ()
- NeuronNICO (double learningRate)
- void calculate ()

  *Calculation of the next output.*

### 5.8.1 Detailed Description

A normalized ICO Neuron that inherit from Neuron or ICO Neuron, respectively.

Learning rule:

$$\Delta\rho_i(t) = \mu \cdot N(u_i(t)) \cdot u_0^{'}(t)$$

where $\mu$ is the learning rate

and $\rho_i$ is the weight

and $u_i$ with $i = 1..n$ are the predictive inputs

and $N()$ is a normalizing function

and $u_0$ is the reflex input.

**learning** rate (default) = 1e-4

**Date:**
01/03/2007 08:28:12 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.8.2 Constructor & Destructor Documentation

#### 5.8.2.1 NeuronNICO::NeuronNICO () `[inline]`

Constructor

```
60 {};
```

#### 5.8.2.2 NeuronNICO::NeuronNICO (double *learningRate*) [inline]

```
62 :NeuronICO(learningRate){};
```

### 5.8.3 Member Function Documentation

#### 5.8.3.1 void NeuronNICO::calculate () [virtual]

Iterates all connected synapses and calculates the **nextoutput_** according to the normalized ICO rule.

**Remarks:**

The flags are:
**0** <=> Reflex input (will not be learned)
**1** <=> Predictive input (will be learned)
**2** <=> Normalized predictive input (will not be learned)

Reimplemented from NeuronICO.

```
57 {
58
59   // Iterate all synapses and sum them up into nextoutput_
60   nextoutput_ = 0;
61   double nextreflex = 0;
62   double normalizedpreflex = 0;
63   std::list<Synapse*>::iterator i, iend = synapsis_.end();
64   for(i = synapsis_.begin(); i!=iend; ++i)
65   {
66     nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
67     if ((*i)->getFlags() == 0)
68     {
69       nextreflex += (*i)->getOutput();
70     }
71
72
73     if ((*i)->getFlags() == 2)
74     {
75       normalizedpreflex = (*i)->getOutput();
76     }
77   }
78
79   if(noLearning_)
80     return;
81
82   // Learn
83   double derivReflex = nextreflex - reflex_;
84
85   for(i = synapsis_.begin(); i!=iend; ++i)
86   {
87     if ((*i)->getFlags() == 1)
88     {
89       (*i)->setWeight((*i)->getWeight() + learningRate_ * derivReflex
90             * normalizedpreflex);
91
92     }
93   }
94   reflex_ = nextreflex;
95 }
```

The documentation for this class was generated from the following files:

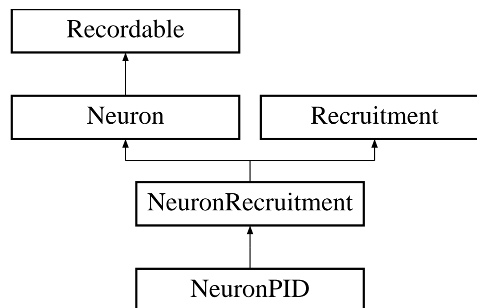
- NeuronNICO.h
- NeuronNICO.cpp

## 5.9 NeuronPID Class Reference

PID Neuron.

`#include <NeuronPID.h>`

Inheritance diagram for NeuronPID::




### Public Member Functions

- NeuronPID ()
- NeuronPID (double setpoint, int timeOffset, double gain, double reset, double deriv)

    *Constructor that sets values.*

- ~NeuronPID ()
- void calculate (double input)

    *Calculation of the next value.*

- void calculate ()

    *Calculates the next output.*

- void setCoefficients (double setpoint, int timeOffset, double gain, double reset, double deriv)

    *Set all values.*

- void reset ()

    *Reseting of the neuron.*

- void setThreshold (double threshold)
- double getThreshold ()

### Private Attributes

- double threshold_
- double error_ [3]
- double buffer_ [MAX_OUTPUT]
- double setpoint_
- int timeOffset_
- double gain_
- double reset_
- double deriv_

### 5.9.1 Detailed Description

An implementation of an usual PID controller. **set-point** (default) = 0

**delay** (default) = 0

$K_p$ (default) = 400

$K_i$ (default) = 2

$K_d$ (default) = 0

**Date:**
01/04/2007 02:43:41 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.9.2 Constructor & Destructor Documentation

### 5.9.2.1 NeuronPID::NeuronPID ()

Constructor

```
35 {
36   reset();
37   setCoefficients(0, 0, 400, 2, 0);
38   setThreshold(0);
39 }
```

### 5.9.2.2 NeuronPID::NeuronPID (double *setpoint*, int *timeOffset*, double *gain*, double *reset*, double *deriv*)

Constructor that sets values.

**Parameters:**
***setpoint*** double - The Set point.
***timeOffset*** int - The Delay.
***gain*** double - The gain (Kp)
***reset*** double - The reset (Ki)
***deriv*** double - The derivative (Kd)

```
53 {
54   this->reset();
55   setCoefficients(setpoint, timeOffset, gain, reset, deriv);
56   setThreshold(0);
57 }
```

### 5.9.2.3 NeuronPID::∼NeuronPID ()

Destructor

```
60 {
61 }
```

## 5.9.3 Member Function Documentation

### 5.9.3.1 void NeuronPID::calculate (double *input*)

Calculation of the next value given a new input. But only if the difference between input and set-point is smaller than **threshold_**

**Parameters:**
***input*** double - A new value.

**Returns:**
-

**Remarks:**

-

```
95 {
96   // If deviation is OK
97   if(fabs(input - setpoint_) < threshold_)
98   {
99     input = setpoint_;
100    }
101
102    // If delay is bad
103    if (timeOffset_ >= MAX_OUTPUT)
104    {
105      exit(1);
106    }
107
108    // Error value shifting
109    error_[2] = error_[1];
110    error_[1] = error_[0];
111    error_[0] = input - setpoint_;
112
113    // PID calculations
114    buffer_[timeOffset_] += gain_ * (+error_[0] - error_[1]) + reset_
115      * error_[0] + deriv_ * (error_[0] - 2 * error_[1] + error_[2]);
116
117    nextoutput_ = buffer_[0];
118
119    // Buffer shifting
120    for (int i = 0; i < timeOffset_; ++i)
121    {
122      buffer_[i] = buffer_[i + 1];
123    }
124
125 }
```

#### 5.9.3.2 void NeuronPID::calculate () `[virtual]`

Calculates the next output. The value provided by the synapses are put into **calculate(double)**

**Returns:**

-

**Remarks:**

-

Reimplemented from Neuron.

```
77 {
78   Neuron::calculate();
79   calculate(nextoutput_);
80 }
```

#### 5.9.3.3 void NeuronPID::setCoefficients (double *setpoint*, int *timeOffset*, double *gain*, double *reset*, double *deriv*)

Set all values.

**Parameters:**

***setpoint*** double - The Set point.

***timeOffset*** int - The Delay.

***gain*** double - The gain (Kp)

***reset*** double - The reset (Ki)

***deriv*** double - The derivative (Kd)

**Returns:**
-

**Remarks:**
-

```
141 {
142   setpoint_ = setpoint;
143   timeOffset_ = timeOffset;
144   gain_ = gain;
145   reset_ = reset;
146   deriv_ = deriv;
147 }
```

#### 5.9.3.4 void NeuronPID::reset () [virtual]

Reseting of the neuron.

**Returns:**
-

**Remarks:**
-

Reimplemented from Neuron.

```
159 {
160   for (int i = 0; i < MAX_OUTPUT; ++i)
161   {
162     buffer_[i] = 0;
163   }
164
165   for (int i = 0; i < 3; ++i)
166   {
167     error_[i] = 0;
168   }
169 }
```

#### 5.9.3.5 void NeuronPID::setThreshold (double *threshold*) [inline]

```
87 {threshold_ = threshold;};
```

#### 5.9.3.6 double NeuronPID::getThreshold () [inline]

```
88 {return threshold_;};
```

### 5.9.4 Member Data Documentation

#### 5.9.4.1 double NeuronPID::threshold_ [private]

#### 5.9.4.2 double NeuronPID::error_[3] [private]

A buffer for the errors

#### 5.9.4.3 double NeuronPID::buffer_[MAX_OUTPUT] [private]

A buffer for the output

#### 5.9.4.4 double NeuronPID::setpoint_ [private]

A set-point

#### 5.9.4.5 int NeuronPID::timeOffset_ [private]

A delay factor

#### 5.9.4.6 double NeuronPID::gain_ [private]

The gain (Kp)

#### 5.9.4.7 double NeuronPID::reset_ [private]

The reset (Ki)

#### 5.9.4.8 double NeuronPID::deriv_ [private]

The derivative (Kd)

The documentation for this class was generated from the following files:

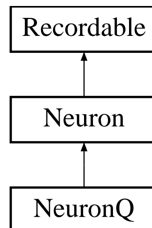
- NeuronPID.h
- NeuronPID.cpp

## 5.10 NeuronQ Class Reference

Q-Learning Neuron.

```
#include <NeuronQ.h>
```

Inheritance diagram for NeuronQ::




### Public Member Functions

- NeuronQ ()
- NeuronQ (double learningRate)

  *Constructor that sets the learning rate.*

- virtual void calculate ()

  *calculation of the next output.*

- void reset ()

  *Reseting of the neuron.*

- void setLearningRate (double learningRate)
- double getLearningRate ()
- void setNoLearning (bool noLearning)
- bool getNoLearning ()

### Private Attributes

- double learningRate_
- bool noLearning_

### 5.10.1 Detailed Description

A Q-Learning Neuron that inherit from Neuron.

Learning rule:

$$\Delta Q(s,a) = \mu \cdot (Q(s',a') - Q(s,a) + r)$$

where $\mu$ is the learning rate

and $s$ is the current state

and $s'$ is the next state

and $a$ is the current action

and $a'$ is the next action

and $r$ is the reward

and $Q()$ are the Q-Values.

**learning** rate (default) = 1e-4

Because you have to put the $Q(s'a')$ into the neuron from outside you can decide if it is the actual next step (SARSA) or the maximized next step (Q-Learning)

**Date:**
01/07/2007 01:15:46 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.10.2 Constructor & Destructor Documentation

### 5.10.2.1 NeuronQ::NeuronQ ()

Constructor

```
47 {
48   reset();
49   learningRate_ = 1e-4;
50   noLearning_ = 0;
51 }
```

### 5.10.2.2 NeuronQ::NeuronQ (double *learningRate*)

Constructor that sets the learning rate.

```
59 {
60   reset();
61   learningRate_ = learningRate;
62   noLearning_ = 0;
63 }
```

## 5.10.3 Member Function Documentation

### 5.10.3.1 void NeuronQ::calculate () [virtual]

calculation of the **nextoutput_** using the sarsa learning scheme.

**Remarks:**
the flags are:
**0** <=> the value of q(s',a')
**1** <=> the state variables
**2** <=> the reward of this state
**3** <=> this neuron lead to the last action
take care of the state values: only the current state has a value of exact 1. the other values could have smaller values, though.

Reimplemented from Neuron.

```
85 {
86   // Iterate all synapses and sum them up into nextoutput_
87   nextoutput_ = 0;
88   double qValue = 0;
89   double action_gate = 0;
90   double reward = 0;
91   std::list<Synapse*>::iterator i, iend = synapsis_.end();
92
93   // neuron specific
94   for(i = synapsis_.begin(); i!=iend; ++i)
95   {
96     // this is the Q-Value of the next step
97     if (((*i)->getFlags()) == 0)
98     {
```

```
99          qValue = (*i)->getOutput();
100       }
101 
102       // reward
103       if (((*i)->getFlags()) == 2)
104       {
105         reward = (*i)->getOutput();
106       }
107 
108       // this action was conducted
109       if (((*i)->getFlags()) == 3)
110       {
111         action_gate = (*i)->getOutput();
112       }
113 
114     }
115 
116     // synapse specific
117     for(i = synapsis_.begin(); i!=iend; ++i)
118     {
119 
120       if (fabs((*i)->getOutput() - 1) < 1e-10)
121       {
122         nextoutput_ = (*i)->getWeight();
123         (*i)->setFactor(action_gate);
124       }
125     }
126 
127 
128 
129 
130     if(noLearning_)
131       return;
132 
133     // Learn
134     for(i = synapsis_.begin(); i!=iend; ++i)
135     {
136       if (((*i)->getOutput()) < 0.55 && ((*i)->getFlags()) == 1)
137       {
138         (*i)->setWeight((*i)->getWeight() + learningRate_ * (*i)->getFactor()
139             * (*i)->getOutput() * (reward + qValue - (*i)->getWeight()));
140       }
141     }
142 
143 }
```

#### 5.10.3.2 void NeuronQ::reset () [virtual]

Reseting of the neuron.

Reimplemented from Neuron.

```
154 {
155   Neuron::reset();
156 
157   std::list<Synapse*>::iterator i, iend = synapsis_.end();
158   for(i = synapsis_.begin(); i!=iend; ++i)
159   {
160     (*i)->setFactor(0);
161   }
162 
163 
164 }
```

#### 5.10.3.3 void NeuronQ::setLearningRate (double *learningRate*) [inline]

```
97 {learningRate_ = learningRate;};
```

#### 5.10.3.4 double NeuronQ::getLearningRate () [inline]

```
98 {return learningRate_;};
```

#### 5.10.3.5 void NeuronQ::setNoLearning (bool *noLearning*) [inline]

```
100 {noLearning_ = noLearning;};
```

#### 5.10.3.6 bool NeuronQ::getNoLearning () [inline]

```
101 {return noLearning_;};
```

### 5.10.4 Member Data Documentation

#### 5.10.4.1 double NeuronQ::learningRate_ [private]

The learning rate

#### 5.10.4.2 bool NeuronQ::noLearning_ [private]

A switch for (no) learning

The documentation for this class was generated from the following files:

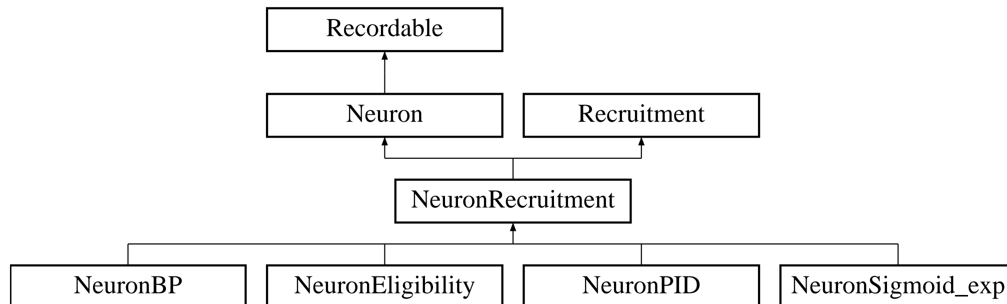
- NeuronQ.h
- NeuronQ.cpp

# 5.11 NeuronRecruitment Class Reference

Recruitment Neuron.

`#include <NeuronRecruitment.h>`

Inheritance diagram for NeuronRecruitment::




## Public Member Functions

- NeuronRecruitment ()
- virtual ∼NeuronRecruitment ()
- void checkFeedback ()

  *Set the fibres according to the feedback.*

### 5.11.1 Detailed Description

A recruitment neuron is a special neuron that additionally check for a proper feedback value which is put into the **fibres** variable.

**See also:**
SynapseRecruitment
Recruitment

**Date:**
01/03/2007 09:16:33 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.11.2 Constructor & Destructor Documentation

#### 5.11.2.1 NeuronRecruitment::NeuronRecruitment () [inline]

Constructor

```
72 {};
```

#### 5.11.2.2 virtual NeuronRecruitment::∼NeuronRecruitment () [inline, virtual]

Destructor

```
75 {};
```

### 5.11.3 Member Function Documentation

#### 5.11.3.1 void NeuronRecruitment::checkFeedback ()

Set the fibres according to the difference of desired value and current output weighted with the feedback factor.

**Returns:**

-

**Remarks:**

The flags are:
**-1** <=> Feedback input (will not be learned and not transmitted)

```
48 {
49   std::list<Synapse*>::iterator i, iend = synapsis_.end();
50   for(i = synapsis_.begin(); i!=iend; ++i)
51   {
52     if((*i)->getFlags() == -1)
53     {
54       setFibres(getFeedbackFactor() * ((*i)->getOutput() - getDesiredValue()));
55     }
56   }
57 }
```

The documentation for this class was generated from the following files:

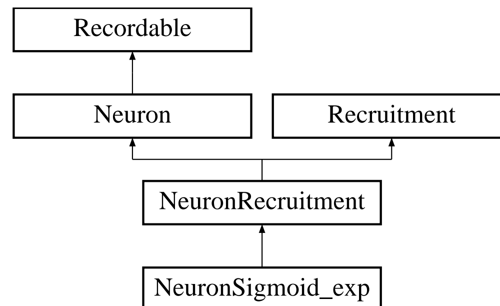
- NeuronRecruitment.h
- NeuronRecruitment.cpp

## 5.12 NeuronSigmoid_exp Class Reference

Sigmoid Neuron (exp).

`#include <NeuronSigmoid_exp.h>`

Inheritance diagram for NeuronSigmoid_exp::




### Public Member Functions

- NeuronSigmoid_exp ()
- NeuronSigmoid_exp (double boost, double threshold)

  *Constructor that sets values.*

- ~NeuronSigmoid_exp ()
- void calculate ()

  *Calculate the next output.*

- void setBoost (double fboost)
- double getBoost ()
- void setThreshold (double fthreshold)
- double getThreshold ()

### Protected Attributes

- double boost_
- double threshold_

### 5.12.1 Detailed Description

An implementation of a sigmoid neuron between 0 and 1:

$$f(x) = \frac{1}{e^{-a(x-b)} - 1}$$

where $a$ is a gain (boost)

and $b$ is a threshold

**boost** (default) = 1

**threshold** (default) = 0

**Date:**
01/04/2007 03:47:29 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.12.2 Constructor & Destructor Documentation

### 5.12.2.1 NeuronSigmoid_exp::NeuronSigmoid_exp ()

Constructor

```
41 {
42   reset();
43   boost_ = 1;
44   threshold_ = 0;
45 }
```

### 5.12.2.2 NeuronSigmoid_exp::NeuronSigmoid_exp (double *boost*, double *threshold*)

Constructor that sets values.

**Parameters:**

***boost*** double - The gain.

***threshold*** double - The threshold.

```
58 {
59   reset();
60   boost_ = boost;
61   threshold_ = threshold;
62 }
```

### 5.12.2.3 NeuronSigmoid_exp::∼NeuronSigmoid_exp ()

Destructor

## 5.12.3 Member Function Documentation

### 5.12.3.1 void NeuronSigmoid_exp::calculate () [virtual]

Calculate the **nextoutput** according to the sigmoid equation.

Reimplemented from Neuron.

```
78 {
79   Neuron::calculate();
80
81   nextoutput_ = 1/(1 + exp(-boost_ * (nextoutput_ - threshold_)) );
82
83 }
```

### 5.12.3.2 void NeuronSigmoid_exp::setBoost (double *fboost*) [inline]

```
106 {boost_ = fboost;};
```

### 5.12.3.3 double NeuronSigmoid_exp::getBoost () [inline]

```
107 {return boost_;};
```

#### 5.12.3.4 void NeuronSigmoid_exp::setThreshold (double *fthreshold*) [inline]

```
108 {threshold_ = fthreshold;};
```

#### 5.12.3.5 double NeuronSigmoid_exp::getThreshold () [inline]

```
109 {return threshold_;};
```

### 5.12.4 Member Data Documentation

#### 5.12.4.1 double NeuronSigmoid_exp::boost_ [protected]

The boost (gain)

#### 5.12.4.2 double NeuronSigmoid_exp::threshold_ [protected]

The threshold

The documentation for this class was generated from the following files:

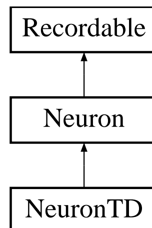
- NeuronSigmoid_exp.h
- NeuronSigmoid_exp.cpp

# 5.13 NeuronTD Class Reference

TD-Learning Neuron.

`#include <NeuronTD.h>`

Inheritance diagram for NeuronTD::




## Public Member Functions

- NeuronTD ()
- NeuronTD (double learningRate)

  *Constructor that sets the learning rate.*

- virtual void calculate ()

  *calculation of the next output.*

- void setLearningRate (double learningRate)
- double getLearningRate ()
- void setNoLearning (bool noLearning)
- bool getNoLearning ()

## Private Attributes

- double learningRate_
- bool noLearning_

### 5.13.1 Detailed Description

A TD-Learning Neuron that inherit from Neuron.

Learning rule:

$$\Delta V(s) = \mu \cdot (V(s') - V(s) + r)$$

where $\mu$ is the learning rate

and $s$ is the current state

and $s'$ is the next state

and $r$ is the reward

and $V()$ are the V-Values.

**learning** rate (default) = 1e-4

**Date:**
01/07/2007 01:15:46 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.13.2 Constructor & Destructor Documentation

### 5.13.2.1 NeuronTD::NeuronTD ()

Constructor

```
41 {
42   reset();
43   learningRate_ = 1e-4;
44   noLearning_ = 0;
45 }
```

### 5.13.2.2 NeuronTD::NeuronTD (double *learningRate*)

Constructor that sets the learning rate.

```
53 {
54   reset();
55   learningRate_ = learningRate;
56   noLearning_ = 0;
57 }
```

## 5.13.3 Member Function Documentation

### 5.13.3.1 void NeuronTD::calculate () [virtual]

calculation of the **nextoutput_** using the sarsa learning scheme.

**Remarks:**

the flags are:
**0** <=> the value of q(s',a')
**1** <=> the state variables
**2** <=> the reward of this state
**3** <=> this neuron lead to the last action
take care of the state values: only the current state has a value of exact 1. the other values could have smaller values, though.

Reimplemented from Neuron.

```
79 {
80   // Iterate all synapses and sum them up into nextoutput_
81   nextoutput_ = 0;
82   double vValue = 0;
83   double reward = 0;
84   std::list<Synapse*>::iterator i, iend = synapsis_.end();
85
86   // neuron specific
87   for(i = synapsis_.begin(); i!=iend; ++i)
88   {
89     // this is the Q-Value of the next step
90     if (((*i)->getFlags()) == 0)
91     {
92       vValue = (*i)->getOutput();
93     }
94
95     // reward
96     if (((*i)->getFlags()) == 2)
97     {
98       reward = (*i)->getOutput();
99     }
100
101   }
```

```
102
103   // synapse specific
104   for(i = synapsis_.begin(); i!=iend; ++i)
105   {
106
107     if (fabs((*i)->getOutput() - 1) < 1e-10)
108     {
109       nextoutput_ = (*i)->getWeight();
110     }
111   }
112
113
114
115
116   if(noLearning_)
117     return;
118
119   // Learn
120   for(i = synapsis_.begin(); i!=iend; ++i)
121   {
122     if (((*i)->getOutput()) < 0.55 && ((*i)->getFlags()) == 1)
123     {
124       (*i)->setWeight((*i)->getWeight() + learningRate_
125           * (*i)->getOutput() * (reward + vValue - (*i)->getWeight()));
126     }
127   }
128
129 }
```

#### 5.13.3.2 void NeuronTD::setLearningRate (double *learningRate*) [inline]

```
84 {learningRate_ = learningRate;};
```

#### 5.13.3.3 double NeuronTD::getLearningRate () [inline]

```
85 {return learningRate_;};
```

#### 5.13.3.4 void NeuronTD::setNoLearning (bool *noLearning*) [inline]

```
87 {noLearning_ = noLearning;};
```

#### 5.13.3.5 bool NeuronTD::getNoLearning () [inline]

```
88 {return noLearning_;};
```

### 5.13.4 Member Data Documentation

#### 5.13.4.1 double NeuronTD::learningRate_ [private]

The learning rate

#### 5.13.4.2 bool NeuronTD::noLearning_ [private]

A switch for (no) learning

The documentation for this class was generated from the following files:

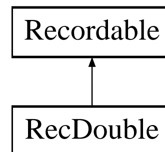
- NeuronTD.h
- NeuronTD.cpp

# 5.14 RecDouble Class Reference

Double-valued Recordable.

`#include <RecDouble.h>`

Inheritance diagram for RecDouble::




## Public Member Functions

- RecDouble ()
- RecDouble (double value)
- double record ()
- void setValue (double fvalue)
- void addValue (double fvalue)
- double getValue ()

## Private Attributes

- double value_

## 5.14.1 Detailed Description

A helpfull class that is used to record double-type values.

This can be done due to the inheritence of the interface Recordable

**See also:**
Recordable
Recorder

**Date:**
01/03/2007 06:33:17 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.14.2 Constructor & Destructor Documentation

### 5.14.2.1 RecDouble::RecDouble () `[inline]`

Constructor

```
70 {};
```

### 5.14.2.2 RecDouble::RecDouble (double *value*) `[inline]`

```
72 :value_(value){};
```

## 5.14.3 Member Function Documentation

### 5.14.3.1 double RecDouble::record () [inline, virtual]

Implements Recordable.

```
79 {return value_;};
```

### 5.14.3.2 void RecDouble::setValue (double *fvalue*) [inline]

```
81 {value_ = fvalue; };
```

### 5.14.3.3 void RecDouble::addValue (double *fvalue*) [inline]

```
82 {value_ += fvalue;};
```

### 5.14.3.4 double RecDouble::getValue () [inline]

```
84 {return record();};
```

## 5.14.4 Member Data Documentation

### 5.14.4.1 double RecDouble::value_ [private]

The value to be recorded

The documentation for this class was generated from the following file:

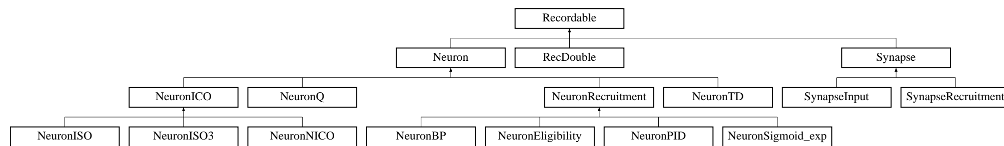
- RecDouble.h

# 5.15 Recordable Class Reference

Interface for Recording.

`#include <Recordable.h>`

Inheritance diagram for Recordable::




## Public Member Functions

- virtual ~Recordable ()
- virtual double record ()=0

## 5.15.1 Detailed Description

An interface that is used to record values.

**See also:**
Recorder
RecDouble

**Date:**
01/03/2007 06:44:53 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.15.2 Constructor & Destructor Documentation

### 5.15.2.1 virtual Recordable::~Recordable () [inline, virtual]

Destructor

```
48 {};
```

## 5.15.3 Member Function Documentation

### 5.15.3.1 virtual double Recordable::record () [pure virtual]

Implemented in Neuron, RecDouble, and Synapse.

The documentation for this class was generated from the following file:

- Recordable.h

## 5.16 Recorder Class Reference

Simple Recorder.

```
#include <Recorder.h>
```

### Public Member Functions

- Recorder (char *filename)
- ~Recorder ()
- void addValue (Recordable *value, std::string const description)

    *Add a value to the recorder with a description.*

- void addValue (Recordable *value)

    *Add a value to the recorder.*

- void init (std::string const mainDescription)

    *Initialize the recording with description.*

- void init ()

    *Initialize the recording.*

- void update ()

    *Write to the file.*

- void update (int Nr, int Max=0, int interval=1000)

    *Write to the file and show the progress.*

### Private Attributes

- std::list< Recordable * > values_
- std::list< std::string > descriptions_
- std::ofstream file_

### 5.16.1 Detailed Description

A simple recorder that takes Recordables (double-typed values) and write them into a specified file.

**See also:**

Recordables
RecDouble

**Date:**

01/03/2007 07:04:23 PM CET

**Version:**

1.0

**Author:**

Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.16.2 Constructor & Destructor Documentation

#### 5.16.2.1 Recorder::Recorder (char * *filename*)

Constructor

```
35 {
36   file_.open(filename);
37 }
```

#### 5.16.2.2 Recorder::∼Recorder ()

Destructor

```
40 {
41   file_.close();
42 }
```

### 5.16.3 Member Function Documentation

#### 5.16.3.1 void Recorder::addValue (Recordable ∗ *value*, std::string const *description*)

Add a value to the recorder with a description.

**See also:**
addValue(Recordable∗)

**Parameters:**
***value*** Recordable∗ - A pointer to a Recordable.
***description*** std::string - A description of the value.

```
153 {
154   values_.push_back(value);
155   descriptions_.push_back(description);
156 }
```

#### 5.16.3.2 void Recorder::addValue (Recordable ∗ *value*)

Add a value to the recorder in the same way as **addValue(Recordable, std::string)** but without a description.

**See also:**
addValue(Recordable∗, std::string)

**Parameters:**
***value*** Recordable∗ - A pointer to a Recordable.

```
172 {
173   addValue(value, "");
174 }
```

#### 5.16.3.3 void Recorder::init (std::string const *mainDescription*)

Initialize the file. Write the descriptions of all values and an additional description.

**Parameters:**
***mainDescription*** std::string - An additional description

**See also:**
init()

```
111 {
112   file_ << "# " << mainDescription << std::endl;
113
114   file_ << "# ";
115
116   std::list<std::string>::iterator i, iend = descriptions_.end();
117   for(i = descriptions_.begin(); i!=iend; ++i)
118   {
119     file_ << " " << (*i) << " |";
120   }
121   file_ << std::endl;
122 }
```

#### 5.16.3.4 void Recorder::init ()

Initialize the file in the same way as **init()** but without an additional description.

**See also:**
init(std::string)

```
136 {
137   init("");
138 }
```

#### 5.16.3.5 void Recorder::update ()

Iterate all values and write them into the file.

```
57 {
58   std::list<Recordable*>::iterator i, iend = values_.end();
59   for(i = values_.begin(); i!=iend; ++i)
60   {
61     file_ << (*i)->record() << "\t";
62   }
63   file_ << std::endl;
64 }
```

#### 5.16.3.6 void Recorder::update (int *Nr*, int *Max* = 0, int *interval* = 1000)

Same as **update()** but with a progress indicator.

**See also:**
update()

**Parameters:**

***Nr*** int - Current position in the loop.

***Max*** int - Number of iterations. (default = 0)
If set, a percentage will be shown.
If not set, the current position will be shown.

***interval*** int - At which interval should the progress be shown (default = 1000)

```
83 {
84   update();
85
86   if(Nr%interval == interval-1)
87   {
88     if(Max)
89       printf("\r%5.2f%%\t ", Nr*100./((double)(Max)));
90     else
91       printf("\r%i\t", Nr);
92
93     fflush(NULL);
94   }
95 }
```

### 5.16.4 Member Data Documentation

#### 5.16.4.1 std::list<Recordable*> Recorder::values_ [private]

A list of values that will be recorded

#### 5.16.4.2 std::list<std::string> Recorder::descriptions_ [private]

A list of descriptions for each recorded value

#### 5.16.4.3 std::ofstream Recorder::file_ [private]

The file that will be written

The documentation for this class was generated from the following files:

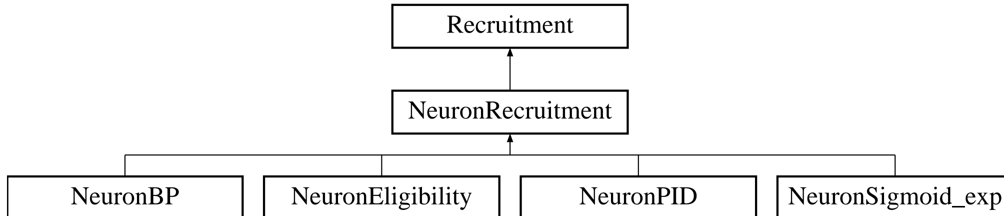
- Recorder.h
- Recorder.cpp

## 5.17 Recruitment Class Reference

Base Recruitment.

`#include <Recruitment.h>`

Inheritance diagram for Recruitment::




### Public Member Functions

- Recruitment ()
- ~Recruitment ()
- void setEntity (double entity)
- double getEntity ()
- void setMaxvalue (double maxvalue)
- double getMaxvalue ()
- void setNrValues (int nrValues)
- int getNrValues ()
- void setNormalizedoutput (double normalizedoutput)
- double getNormalizedoutput ()
- void setIsRecruitment (bool isRecruitment)
- bool getIsRecruitment ()
- void setFibres (double fibres)
- double getFibres ()
- void setDesiredValue (double desiredValue)
- double getDesiredValue ()
- void setFeedbackFactor (double feedbackFactor)
- double getFeedbackFactor ()
- RecDouble * recOutput ()
- double record ()

### Private Attributes

- double entity_
- double maxvalue_
- int nrValues_
- bool isRecruitment_
- RecDouble normalizedoutput_
- double desiredValue_
- double feedbackFactor_
- double fibres_

### 5.17.1 Detailed Description

The recruitment mechanism can adopt to different external circumstances as long as they behave in first order linear.

For this you need a smallest **entity**, a **maximal value** and a consistent **number** of entities the mechanism consist of.

**entity** (default) = 1

**max**. **value** (default) = 10

**nr**. **values** (default) = 10

**isRecruitment** (default) = false

One possibility to use the recruitment mechanism is to predict the upcoming scale of behavior. If you 'know' how heavy a cup of water you just put this 'knowledge' into the mechanism.

But if your prediction was wrong or you don't have any prediction at all you need a feedback mechanism that is also provided. A **desired value** is achieved by activating sufficient **fibres** according to the difference of the **desired output** and the **current output** weighted by a gain called **feedback factor**.

**desired value** (default) = 0

**feedback factor** (default) = 1

**fibres** (default) = 0

**See also:**
NeuronRecruitment
SynapseRecruitment

**Date:**
01/03/2007 08:57:05 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.17.2 Constructor & Destructor Documentation

### 5.17.2.1 Recruitment::Recruitment () [inline]

Constructor

```
74                  :entity_(1.),maxvalue_(10.),nrValues_(10),isRecruitment_(false),desiredValue_(0),feedb
75      {normalizedoutput_.setValue(0.);};
```

### 5.17.2.2 Recruitment::∼Recruitment () [inline]

Destructor

```
78 {};
```

## 5.17.3 Member Function Documentation

### 5.17.3.1 void Recruitment::setEntity (double *entity*) [inline]

```
86 {entity_ = entity;};
```

### 5.17.3.2 double Recruitment::getEntity () [inline]

```
87 {return entity_;};
```

### 5.17.3.3 void Recruitment::setMaxvalue (double *maxvalue*) [inline]

```
89 {maxvalue_ = maxvalue;};
```

### 5.17.3.4 double Recruitment::getMaxvalue () [inline]

```
90 {return maxvalue_;};
```

#### 5.17.3.5 void Recruitment::setNrValues (int *nrValues*) [inline]

```
92 {nrValues_ = nrValues;};
```

#### 5.17.3.6 int Recruitment::getNrValues () [inline]

```
93 {return nrValues_;};
```

#### 5.17.3.7 void Recruitment::setNormalizedoutput (double *normalizedoutput*) [inline]

```
96         {normalizedoutput_.setValue(normalizedoutput);};
```

#### 5.17.3.8 double Recruitment::getNormalizedoutput () [inline]

```
97 {return normalizedoutput_.getValue();};
```

#### 5.17.3.9 void Recruitment::setIsRecruitment (bool *isRecruitment*) [inline]

```
99 {isRecruitment_ = isRecruitment;};
```

#### 5.17.3.10 bool Recruitment::getIsRecruitment () [inline]

```
100 {return isRecruitment_;};
```

#### 5.17.3.11 void Recruitment::setFibres (double *fibres*) [inline]

```
102 {fibres_ = fibres;};
```

#### 5.17.3.12 double Recruitment::getFibres () [inline]

```
103 {return fibres_;};
```

#### 5.17.3.13 void Recruitment::setDesiredValue (double *desiredValue*) [inline]

```
105 {desiredValue_ = desiredValue;};
```

#### 5.17.3.14 double Recruitment::getDesiredValue () [inline]

```
106 {return desiredValue_;};
```

#### 5.17.3.15 void Recruitment::setFeedbackFactor (double *feedbackFactor*) [inline]

```
109         {feedbackFactor_ = feedbackFactor;};
```

#### 5.17.3.16 double Recruitment::getFeedbackFactor () [inline]

```
110 {return feedbackFactor_;};
```

#### 5.17.3.17 RecDouble* Recruitment::recOutput () [inline]

```
112 {return &normalizedoutput_;};
```

#### 5.17.3.18 double Recruitment::record () [inline]

```
113 {return getNormalizedoutput();}
```

### 5.17.4 Member Data Documentation

#### 5.17.4.1 double Recruitment::entity_ [private]

The basic entity. The smallest value possible.

#### 5.17.4.2 double Recruitment::maxvalue_ [private]

The biggest value possible

#### 5.17.4.3 int Recruitment::nrValues_ [private]

The number of entities needed to reach **maxvalue_**

#### 5.17.4.4 bool Recruitment::isRecruitment_ [private]

A switch for Recruitment

#### 5.17.4.5 RecDouble Recruitment::normalizedoutput_ [private]

The normalized output

#### 5.17.4.6 double Recruitment::desiredValue_ [private]

The desired Value

#### 5.17.4.7 double Recruitment::feedbackFactor_ [private]

The feedback factor (gain)

#### 5.17.4.8 double Recruitment::fibres_ [private]

The number of fibres needed to reach the **desiredValue_** with the current output weighted by the **feedbackFactor_**

The documentation for this class was generated from the following file:

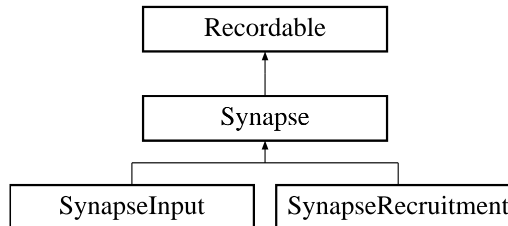
- Recruitment.h

## 5.18 Synapse Class Reference

Basic Synapse.

`#include <Synapse.h>`

Inheritance diagram for Synapse::




### Public Member Functions

- Synapse ()
- Synapse (Neuron *neuron)

  *Constructor that attaches a neuron.*

- virtual ~Synapse ()
- void setNeuron (Neuron *neuron)

  *Attach the synapse to a neuron.*

- virtual double getOutput ()
- void setWeight (double weight)
- double getWeight ()
- void setFlags (int flags)
- int getFlags ()
- void setFactor (double factor)
- double getFactor ()
- RecDouble * recWeight ()
- double record ()

### Protected Attributes

- Neuron * neuron_
- RecDouble weight_
- int flags_
- double factor_

### 5.18.1 Detailed Description

This is the base synapse all other synapse inherit from. Basically it has a weight, a flag and an output.

**weight(0)** = 1

**flag(0)** = 0

These properties will not be interpreted by the synapse but by the neuron. Thus the output from the synapse is not weighted.

The synapse itself inherits an interface called Recordable. Because of that you can easily record the weight of all synapses during learning.

**See also:**

RecDouble
Recordable
Recorder

**Date:**

01/03/2007 05:05:29 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

## 5.18.2 Constructor & Destructor Documentation

### 5.18.2.1 Synapse::Synapse ()

Constructor

```
47 {
48   setWeight(0.);
49   setFlags(0);
50   setFactor(1.);
51 }
```

### 5.18.2.2 Synapse::Synapse (Neuron * *neuron*)

Constructor that attaches a neuron.

**Parameters:**
***neuron*** Neuron* - A pointer to a neuron.

```
61 {
62   neuron_ = neuron;
63   setWeight(0.);
64   setFlags(0);
65   setFactor(1.);
66 }
```

### 5.18.2.3 Synapse::~Synapse () [virtual]

Destructor

```
69 {
70 }
```

## 5.18.3 Member Function Documentation

### 5.18.3.1 void Synapse::setNeuron (Neuron * *neuron*)

Attach the synapse to a neuron.

**Parameters:**
***neuron*** Neuron* - A pointer to a neuron

```
88 {
89   neuron_ = neuron;
90 }
```

### 5.18.3.2 virtual double Synapse::getOutput () [inline, virtual]

Reimplemented in SynapseInput, and SynapseRecruitment.

```
88 {return neuron_->getOutput();};
```

#### 5.18.3.3 void Synapse::setWeight (double *weight*) [inline]

```
90 {weight_.setValue(weight);};
```

#### 5.18.3.4 double Synapse::getWeight () [inline]

```
91 {return weight_.getValue();};
```

#### 5.18.3.5 void Synapse::setFlags (int *flags*) [inline]

```
93 {flags_ = flags;};
```

#### 5.18.3.6 int Synapse::getFlags () [inline]

```
94 {return flags_;};
```

#### 5.18.3.7 void Synapse::setFactor (double *factor*) [inline]

```
96 {factor_ = factor;};
```

#### 5.18.3.8 double Synapse::getFactor () [inline]

```
97 {return factor_;};
```

#### 5.18.3.9 RecDouble* Synapse::recWeight () [inline]

```
99 {return &weight_;};
```

#### 5.18.3.10 double Synapse::record () [inline, virtual]

Implements Recordable.

```
100 {return getOutput();}
```

### 5.18.4 Member Data Documentation

#### 5.18.4.1 Neuron* Synapse::neuron_ [protected]

The attached neuron

#### 5.18.4.2 RecDouble Synapse::weight_ [protected]

The according weight

#### 5.18.4.3 int Synapse::flags_ [protected]

An arbitrary usable flag

#### 5.18.4.4 double Synapse::factor_ [protected]

A factor that can be used to scale the weight

The documentation for this class was generated from the following files:

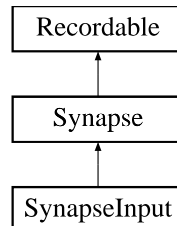
- Synapse.h
- Synapse.cpp

## 5.19 SynapseInput Class Reference

Sensory (Input) Synapse.

`#include <SynapseInput.h>`

Inheritance diagram for SynapseInput::




### Public Member Functions

- SynapseInput ()
- void setOutput (double foutput_)
- double getOutput ()

### Private Attributes

- double output_

### 5.19.1 Detailed Description

A sensory synapse that gets it input not from other neurons but from the external world.

**Date:**
01/03/2007 06:23:15 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.19.2 Constructor & Destructor Documentation

#### 5.19.2.1 SynapseInput::SynapseInput ()

Constructor

```
33 {
34   Synapse::setWeight(1.);
35 }
```

### 5.19.3 Member Function Documentation

#### 5.19.3.1 void SynapseInput::setOutput (double *foutput_*) [inline]

```
56 {output_ = foutput_;};
```

#### 5.19.3.2 double SynapseInput::getOutput () [inline, virtual]

Reimplemented from Synapse.

```
57 {return output_;};
```

## 5.19.4 Member Data Documentation

#### 5.19.4.1 double SynapseInput::output_ [private]

Since there is no attached neuron the synapse have to have its own input

The documentation for this class was generated from the following files:

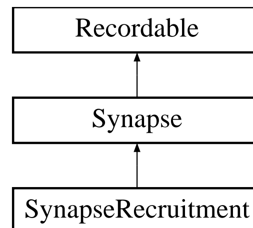
- SynapseInput.h
- SynapseInput.cpp

# 5.20 SynapseRecruitment Class Reference

Recruitment Synapse.

`#include <SynapseRecruitment.h>`

Inheritance diagram for SynapseRecruitment::




## Public Member Functions

- SynapseRecruitment ()
- SynapseRecruitment (NeuronRecruitment ∗neuronRecruitment, Neuron ∗neuron)

  *Constructor that attaches neurons.*

- ∼SynapseRecruitment ()
- double getOutput ()

## Private Attributes

- NeuronRecruitment ∗ neuronRecruitment_

### 5.20.1 Detailed Description

A recruitment synapse is a special synapse that additionally is connected to a recruitment neuron.

**See also:**
NeuronRecruitment
Recruitment

**Date:**
01/03/2007 09:07:03 PM CET

**Version:**
1.0

**Author:**
Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de BCCN Goettingen

### 5.20.2 Constructor & Destructor Documentation

#### 5.20.2.1 SynapseRecruitment::SynapseRecruitment ()

Constructor

```
36 {
37 }
```

#### 5.20.2.2 SynapseRecruitment::SynapseRecruitment (NeuronRecruitment * *neuronRecruitment*, Neuron * *neuron*)

Constructor that attaches both a neuron and a recruitment neuron.

**Parameters:**

***neuronRecruitment*** NeuronRecruitment* - A pointer to a recruitment neuron

***neuron*** Neuron* - A pointer to a neuron

```
51 {
52   neuronRecruitment_ = neuronRecruitment;
53   neuron_ = neuron;
54 }
```

#### 5.20.2.3 SynapseRecruitment::~SynapseRecruitment ()

Destructor

### 5.20.3 Member Function Documentation

#### 5.20.3.1 double SynapseRecruitment::getOutput () [inline, virtual]

Reimplemented from Synapse.

```
75 {return neuronRecruitment_->getNormalizedoutput();};
```

### 5.20.4 Member Data Documentation

#### 5.20.4.1 NeuronRecruitment* SynapseRecruitment::neuronRecruitment_ [private]

An additional recruitment neuron

The documentation for this class was generated from the following files:

- SynapseRecruitment.h
- SynapseRecruitment.cpp

# Chapter 6

# Neuronal Network Simulator File Documentation

## 6.1 Neuron.cpp File Reference

```
/** Basic Neuron.
 *
 *           \class  Neuron
 *
 *                   This is the base neuron all other neurons inherit
 *                   from. Basically it sums up the weighted values of
 *                   all connected synapses.\n
 *                   First this summation is calculated for the whole
 *                   network followed by an updating step.
 *
 *            \date  12/20/2006 02:02:44 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "Neuron.h"
#include "Synapse.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// Constructor and Destructor
// =====================================================================================
Neuron::Neuron ()
{
}


Neuron::~Neuron ()
{
  clear();
```

```
}


// ==================================================================================
// ==================================================================================


/** Add a synapse to this neuron.
 *
 *              Add a synapse to this neuron
 *
 *      @param  synapse Synapse* - A pointer to a synapse that will be connected
 *     @return   -
 *
 *    @remarks   -
 *
 */
void Neuron::addSynapse(Synapse* synapse)
{
  synapsis_.push_back(synapse);
}


/** Reset the neuron to zero.
 *
 *              Variable \b output_ and \b nextoutput_ are set 0
 *
 *     @return   -
 *
 *    @remarks   -
 *
 */
void Neuron::reset()
{
  output_ = 0;
  nextoutput_ = 0;
}


/** Calculation of the next output
 *
 *              Iterating the connected synapses and summing up
 *              their output weighted with their weight
 *              The output is stored into the variable \b nextoutput_
 *
 *     @return   -
 *
 *    @remarks  Use always the variable \b nextoutput_
 */
void Neuron::calculate()
{
  nextoutput_ = 0;
  std::list<Synapse*>::iterator i, iend = synapsis_.end();
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
  }
}

/** Updating after calculation
 *
 *               After all neurons within the Neuronal Network are
 *               calculated they become updated
 *               \b nextoutput_ -> \b output_
 *
 *     @return   -
 *
```

```
 *    @remarks   -
 */
void Neuron::update()
{
  output_ = nextoutput_;
}


/** Destruction of the neuron
 *
 *              Destruction of all connected synapses and the
 *              neuron itself
 *
 *     @return   -
 *
 *    @remarks  The complete neural network destruct itself
 *              recursive thus you don't have to do this by hand
 *
 */
void Neuron::clear()
{
  nextoutput_ = 0;
  std::list<Synapse*>::iterator i, imax = synapsis_.end();
  for(i = synapsis_.begin(); i != imax; ++i)
    delete (*i);
  synapsis_.clear();
}

// ====================================================================================
// ====================================================================================
```

## 6.2 Neuron.h File Reference

```
/** Basic Neuron.
 *
 *       \filename  Neuron.h
 *
 *          \class  Neuron
 *
 *                  This is the base neuron all other neurons inherit
 *                  from. Basically it sums up the weighted values of
 *                  all connected synapses.\n
 *                  First this summation is calculated for the whole
 *                  network followed by an updating step.
 *
 *           \date  12/20/2006 02:02:44 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

#ifndef _NEURON_H_
#define _NEURON_H_

// =====================================================================================
// System Includes
// =====================================================================================

#include <list>
#include <iostream>

// =====================================================================================
// Project Includes
// =====================================================================================

#include "Recordable.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================

class Synapse;


// =====================================================================================
// =====================================================================================
class Neuron : public Recordable
{

  public:

    // ====================  LIFECYCLE   ========================================

    /*! Constructor */
    Neuron ();

    /*! Destructor */
    virtual ~Neuron ();

    // ====================  OPERATORS   ========================================

    // ====================  OPERATIONS  ========================================

    /** Add a synapse to this neuron.
     *
     *              Add a synapse to this neuron
```

```
     *
     *      @param  synapse Synapse* - A pointer to a synapse that will be connected
     *
     */
    void addSynapse(Synapse* synapse);

    /** Reset the neuron to zero.
     *
     * Variable \b output_ and \b nextoutput_ are set 0
     *
     */
    virtual void reset();

    /** Calculation of the next output
     *
     *              Iterating the connected synapses and summing up
     *              their output weighted with their weight
     *              The output is stored into the variable \b nextoutput_
     *
     *    @remarks  Use always the variable \b nextoutput_
     */
    virtual void calculate();

    /** Updating after calculation
     *
     *               After all neurons within the Neuronal Network are
     *               calculated they become updated
     *               \b nextoutput_ -> \b output_
     *
     */
    virtual void update();

    /** Destruction of the neuron
     *
     *              Destruction of all connected synapses and the
     *              neuron itself
     *
     *    @remarks  The complete neural network destruct itself
     *              recursive thus you don't have to do this by hand
     *
     */
    void clear();

    // ====================  ACCESS      =========================================

    void setOutput(double output) {output_ = output;};
    double getOutput() {return output_;};

    double record(){return output_;}

    // ====================  INQUIRY     =========================================

  protected:
    /*! A list of synapses */
    std::list<Synapse*> synapsis_;

    /*! The output of the neuron */
    double output_;

    /*!
     *  The next output of the neuron that is determined during the
     *  calculating step and set to output during the update step
     */
    double nextoutput_;

  private:
```

```
};

#endif
```

## Classes

- class Neuron
  *Basic Neuron.*

## 6.3 NeuronalNet.cpp File Reference

```
/** Basic Neuronal Network that contains all compartments.
 *
 *           \class  NeuronalNet
 *
 *                   All compartments (Neurons and Synapses) are put
 *                   into the Neuronal Network. \n
 *                   During the calculation the next output of each
 *                   neuron is computed. \n
 *                   Afterwards during the update phase all calculated
 *                   values are set to the output of the neurons.
 *
 *            \date  01/03/2007 03:49:12 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronalNet.h"
#include "Neuron.h"
#include "Synapse.h"
#include "SynapseInput.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// Constructor and Destructor
// =====================================================================================
NeuronalNet::NeuronalNet ()
{
}

NeuronalNet::~NeuronalNet ()
{
  clear();
}


// =====================================================================================
// =====================================================================================


/** Reset the Neuronal Network.
 *
 *              Iterate all neurons and call their reset function
 *
 *      @return   -
 *
 *     @remarks   -
 */
void NeuronalNet::reset()
{
  std::list<Neuron*>::iterator i, iend = neurons_.end();
  for(i = neurons_.begin(); i!=iend; ++i)
  {
    (*i)->reset();
  }
```

```
}

/** Compute and update all neurons.
 *
 *              Iterate all neurons. First call their \b calculate()
 *              method and after all the output values were
 *              computed call the \b update() method.
 *
 *     @return   -
 *
 *    @remarks  Mostly the \b update() method only sets the value of
 *              the output to the calculated value
 */
void NeuronalNet::update()
{
  std::list<Neuron*>::iterator i, iend = neurons_.end();
  for(i = neurons_.begin(); i!=iend; ++i)
  {
    (*i)->calculate();
  }
  for(i = neurons_.begin(); i!=iend; ++i)
  {
    (*i)->update();
  }
}

/** Add a neuron to the Neuronal Network.
 *
 *              Add a neuron to the Neuronal Network
 *
 *      @param  neuron Neuron* - A pointer to a neuron that will be
 *                               calculated
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronalNet::addNeuron(Neuron* neuron)
{
  neurons_.push_back(neuron);
}

/** Add a neuron as an input to the Neuronal Network.
 *
 *              Add a neuron as an input to the Neuronal Network
 *
 *      @param  neuron Neuron* - A pointer to a neuron that will be
 *                               calculated
 *     @return   -
 *
 *    @remarks  The input to this neuron is set from outside.
 *    @remarks  A neuron should only be added either as input or as an
 *              usual neuron.
 *
 *        @see  addNeuron
 */
void NeuronalNet::setInput(Neuron* neuron)
{
  addNeuron(neuron);
  Synapse* s = new SynapseInput();
  inputs_.push_back((SynapseInput*)(s));
  neuron->addSynapse(s);
}

/** Add a neuron as an output to the Neuronal Network.
 *
 *              Add a neuron as an output to the Neuronal Network
 *
```

```
 *      @param  neuron Neuron* - A pointer to a neuron that will be
 *                              calculated
 *     @return   -
 *
 *    @remarks  The output can be easily used outside
 *    @remarks  A neuron should additionally added as an output.
 *
 *        @see  addNeuron
 */
void NeuronalNet::setOutput(Neuron* neuron)
{
  outputs_.push_back(neuron);
}

/** Set the input of the Neuronal Network.
 *
 *              Set the input values of all input neurons that are
 *              within the Neuronal Network.
 *
 *      @param  values[] double - An array that has the same size as
 *                               input neurons embedded
 *     @return   -
 *
 *    @remarks  There is no size check! You have to take by yourself.
 */
void NeuronalNet::setInputValues(double values[])
{
  double* cur = values;
  std::list<SynapseInput*>::iterator i, iend = inputs_.end();
  for(i = inputs_.begin(); i!=iend; ++i)
  {
    (*i)->setOutput(*cur);
    cur++;
  }

}

/** Get the output of the Neuronal Network
 *
 *              Get the output values of all neurons that were
 *              assigned as output neurons.
 *
 *     @return  double* - A pointer with all output values stored
 *
 *    @remarks  There is no information about the size of the array!
 *              You have to rememeber by yourself
 */
double* NeuronalNet::getOutputValues()
{
  double* out = new double[outputs_.size()];
  double* cur = out;

  std::list<Neuron*>::iterator i, iend = outputs_.end();
  for(i = outputs_.begin(); i!=iend; ++i)
  {
    *cur = (*i)->getOutput();
    cur++;
  }

  return out;
}

/** Destruction of the Neuronal Network.
 *
 *              Iterate all neurons and call the \b clear() method.
 *
 *     @return   -
```

```
 *
 *    @remarks  The complete neural network destruct itself
 *              recursive thus you don't have to do this by hand
 */
void NeuronalNet::clear()
{

  std::list<Neuron*>::iterator i, imax = neurons_.end();
  for(i = neurons_.begin(); i != imax; ++i)
    delete (*i);
  neurons_.clear();
  inputs_.clear();
}
```

## 6.4 NeuronalNet.h File Reference

```
/** Basic Neuronal Network that contains all compartments.
 *
 *        \filename  NeuronalNet.h
 *
 *           \class  NeuronalNet
 *
 *                   All compartments (Neurons and Synapses) are put
 *                   into the Neuronal Network. \n
 *                   During the calculation the next output of each
 *                   neuron is computed. \n
 *                   Afterwards during the update phase all calculated
 *                   values are set to the output of the neurons.
 *
 *            \date  01/03/2007 03:49:12 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */


#ifndef _NeuronalNet_h_
#define _NeuronalNet_h_

// =====================================================================================
// System Includes
// =====================================================================================

#include <iostream>
#include <list>

// =====================================================================================
// Project Includes
// =====================================================================================


// =====================================================================================
// Forward class declarations
// =====================================================================================

class Neuron;
class SynapseInput;


// =====================================================================================
// =====================================================================================
class NeuronalNet
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    NeuronalNet ();

    /*! Destructor */
    ~NeuronalNet ();

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    /** Add a neuron to the Neuronal Network.
```

```
 *
 *              Add a neuron to the Neuronal Network
 *
 *      @param  neuron Neuron* - A pointer to a neuron that will be
 *                               calculated
 */
void addNeuron(Neuron* neuron);

/** Add a neuron as an output to the Neuronal Network.
 *
 *              Add a neuron as an output to the Neuronal Network
 *
 *      @param  neuron Neuron* - A pointer to a neuron that will be
 *                               calculated
 *
 *    @remarks  The output can be easily used outside
 *    @remarks  A neuron should additionally added as an output.
 *
 *        @see  addNeuron
 */
void setOutput(Neuron* neuron);

/** Add a neuron as an input to the Neuronal Network.
 *
 *              Add a neuron as an input to the Neuronal Network
 *
 *      @param  neuron Neuron* - A pointer to a neuron that will be
 *                               calculated
 *
 *    @remarks  The input to this neuron is set from outside.
 *    @remarks  A neuron should only be added either as input or as an
 *              usual neuron.
 *
 *        @see  addNeuron
 */
void setInput(Neuron* neuron);


/** Set the input of the Neuronal Network.
 *
 *              Set the input values of all input neurons that are
 *              within the Neuronal Network.
 *
 *      @param  values[] double - An array that has the same size as
 *                                input neurons embedded
 *
 *    @remarks  There is no size check! You have to take by yourself.
 */
void setInputValues(double values[]);

/** Get the output of the Neuronal Network
 *
 *              Get the output values of all neurons that were
 *              assigned as output neurons.
 *
 *     @return  double* - A pointer with all output values stored
 *
 *    @remarks  There is no information about the size of the array!
 *              You have to rememeber by yourself
 */
double* getOutputValues();


/** Compute and update all neurons.
 *
 *              Iterate all neurons. First call their \b calculate()
 *              method and after all the output values were
```

```
     *              computed call the \b update() method.
     *
     *    @remarks  Mostly the \b update() method only sets the value of
     *              the output to the calculated value.
     */
    void update();

    /** Reset the Neuronal Network.
     *
     *              Iterate all neurons and call their reset function.
     *
     */
    void reset();

    /** Destruction of the Neuronal Network.
     *
     *              Iterate all neurons and call the \b clear() method.
     *
     *    @remarks  The complete neural network destruct itself
     *              recursive thus you don't have to do this by hand.
     */
    void clear();

    // ====================  ACCESS      =========================================

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! A list of all neurons except the input neurons */
    std::list<Neuron*> neurons_;

    /*! A list of all input neurons */
    std::list<SynapseInput*> inputs_;

    /*! A list of all neurons that are outputs */
    std::list<Neuron*> outputs_;
};

#endif
```

## Classes

- class NeuronalNet

  *Basic Neuronal Network that contains all compartments.*

## 6.5 NeuronBP.cpp File Reference

```
/** Bandpass Neuron
 *
 *          \class  NeuronBP
 *
 *                  A bandpass neuron can emulate a filter
 *                  according to:\n
 *                  \f[
 *                  h(t)=\frac{1}{b} e^{at} \sin{(bt)}
 *                  \f]
 *                  with \f$a=\frac{\pi f}{q}\f$
 *                  and \f$b=\sqrt{(2 \pi f)^2-a^2}\f$ yields
 *                  into:
 *                  \f[
 *                  y[(n+1)T] =
 *                  -2 \cdot e^{-a T} cos{b T} y[nY] -
 *                  e^{-2 \cdot a T} y[(n-1)T ]+
 *                  \frac{1}{b}e^{aT}\sin{(bT)} x[nT]
 *                  \f]
 *                  The last factor \f$x[n T]\f$ is set to 1
 *                  (irrelevant because of the normalization) and T
 *                  is set to 1, too.\n
 *
 *                  \b f (default) = 0.1 \n
 *                  \b q (default) = 0.51 \n
 *                  \b normlize (default) = true \n
 *
 *                  The higher the frequency the sharper the peak\n
 *                  The higher the quality the more is the filter
 *                  oscillating
 *
 *           \date  01/03/2007 09:26:11 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "Neuron.h"
#include "Synapse.h"
#include "NeuronBP.h"

// =====================================================================================
// defines and consts
// =====================================================================================

#define max(a,b) (((a) > (b)) ? (a) : (b))

// =====================================================================================
// Constructor and Destructor
// =====================================================================================

/** Constructor that sets frequency and quality.
 *
 *             Constructor that sets frequency and quality.
 *
 *      @param  f double - The frequency.
 *      @param  q double - The quality.
 *
 */
NeuronBP::NeuronBP(double f, double q)
```

```
{
  reset();
  normalize_ = true;
  setFQ(f,q);
}


// ========================================================================================
// ========================================================================================

/** Initialization
 *
 *              Initialization of the \b denomiator_ needed for the
 *              calculation of the filter.
 *
 *      @param  f double - The frequency
 *      @param  q double - The quality
 *     @return   -
 *
 *    @remarks  The quality should be bigger than 0.51
 */
void NeuronBP::setFQ(double f, double q)
{
  // If Q is ok
  if (q > 0)
  {

    double fTimesPi = f * M_PI * 2.;
    double e = fTimesPi / (2. * q);

    // If root is ok
    if ((fTimesPi * fTimesPi - e * e) > 0)
    {

      double w = sqrt(fTimesPi * fTimesPi - e * e);
      denominator_[0] = -2. * exp(-e) * cos(w);
      denominator_[1] = exp(-2. * e);

      calcNorm();

    }
    // If root is bac
    else
    {
      std::cerr << "ERROR - bandpass: The quality is too low" << std::endl;
      exit(1);
    }

  }
  // If Q is bad
  else
  {
    std::cerr << "bandpass: The quality is rubbish!" << std::endl;
    exit(1);
  }
}

/** Calculation of the normalizing factor.
 *
 *              Calculation of the normalizing factor.
 *
 *     @return   -
 *
 *    @remarks  The 'search' for the maximum is limited to 200. This
 *              can cause trouble if the frequency is to low.
 */
void NeuronBP::calcNorm()
```

```
{
  norm_ = 1.;
  // Reset of buffer
  for (int i = 0; i < 2; i++)
  {
    buffer_[i] = 0;
  }
  // Calculation of new norm
  double nnorm = 0;
  for (int i = 0; i < 200; i++)
  {
    calculate((i == 5) ? 1 : 0);
    nnorm = max(nnorm, nextoutput_);
  }
  nextoutput_ = 0;

  if (nnorm != 0)
    norm_ = nnorm;

  // Reset of buffer
  for (int i = 0; i < 2; i++)
  {
    buffer_[i] = 0;
  }
}

/** Calculation of the next value
 *
 *             Calculation of the next value according to the filter
 *             equations.
 *
 *      @param  in double - The new value.
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronBP::calculate(double in)
{

  nextoutput_ = in
              - denominator_[0] * buffer_[0]
              - denominator_[1] * buffer_[1];

  buffer_[1] = buffer_[0];
  buffer_[0] = nextoutput_;


  if (normalize_)
  {
    nextoutput_ /= norm_;
  }


}

/** Calculation of the next output.
 *
 *              Calculation of the \b nextoutput_ either without
 *              recruitment or with recruitment:
 *              no recruitment: The value provided by the connected
 *              synapses are put into the filter as a new value
 *              recruitment: The value provided by the connected
 *              synapses are put into the recruitment mechanism (\b
 *              fibre) and only the entity is put into the filter as
 *              a new value.
 *
 *     @return   -
```

```
 *
 *    @remarks   -
 */
void NeuronBP::calculate()
{
  Neuron::calculate();

  if(getIsRecruitment() && nextoutput_)
  {
    setFibres(nextoutput_);
    calculate(getEntity());
  }
  else
  {
    calculate(nextoutput_);
  }

}

/** Updating of the neuron.
 *
 *              Updating of the neuron either without recruitment
 *              (nothing changed compared to Neuron) or with
 *              recruitment: The calculated value is put as the
 *              normalized one into the learning pathway (\b
 *              normalized \b output) and the output weighted with the
 *              fibres is set as 'real' output. \n
 *              Because of the feedback the number of used fibres may
 *              have changed.
 *
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronBP::update()
{
  Neuron::update();

  setNormalizedoutput(output_);
  if(getIsRecruitment())
  {
    checkFeedback();
    output_ *= getFibres();
  }

}

/** Reseting of the neuron.
 *
 *             Reseting of the neuron.
 *
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronBP::reset()
{
  buffer_[0] = 0;
  buffer_[1] = 0;
  Neuron::reset();
  output_ = 0;
  nextoutput_ = 0;
}
```

## 6.6 NeuronBP.h File Reference

```
/** Bandpass Neuron
 *
 *          \class  NeuronBP
 *
 *                  A bandpass neuron can emulate a filter
 *                  according to:\n
 *                  \f[
 *                  h(t)=\frac{1}{b} e^{at} \sin{(bt)}
 *                  \f]
 *                  with \f$a=\frac{\pi f}{q}\f$
 *                  and \f$b=\sqrt{(2 \pi f)^2-a^2}\f$ yields
 *                  into:
 *                  \f[
 *                  y[(n+1)T] =
 *                  -2 \cdot e^{-a T} cos{b T} y[nY] -
 *                  e^{-2 \cdot a T} y[(n-1)T ]+
 *                  \frac{1}{b}e^{aT}\sin{(bT)} x[nT]
 *                  \f]
 *                  The last factor \f$x[n T]\f$ is set to 1
 *                  (irrelevant because of the normalization) and T
 *                  is set to 1, too.\n
 *
 *                  \b f (default) = 0.1 \n
 *                  \b q (default) = 0.51 \n
 *                  \b normlize (default) = true \n
 *
 *                  The higher the frequency the sharper the peak\n
 *                  The higher the quality the more is the filter
 *                  oscillating
 *
 *           \date  01/03/2007 09:26:11 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

#ifndef _NeuronBP_h_
#define _NeuronBP_h_

// ===================================================================================
// System Includes
// ===================================================================================

#include <iostream>
#include <cmath>

// ===================================================================================
// Project Includes
// ===================================================================================

#include "NeuronRecruitment.h"

// ===================================================================================
// Forward class declarations
// ===================================================================================


// ===================================================================================
// ===================================================================================
class NeuronBP : public NeuronRecruitment
{

  public:
```

```
// ====================  LIFECYCLE   ========================================

/*! Constructor */
NeuronBP(double f=DEF_F, double q=DEF_Q);

// ====================  OPERATORS   ========================================

// ====================  OPERATIONS  ========================================

/** Calculation of the next output.
 *
 *              Calculation of the \b nextoutput_ either without
 *              recruitment or with recruitment:
 *              no recruitment: The value provided by the connected
 *              synapses are put into the filter as a new value
 *              recruitment: The value provided by the connected
 *              synapses are put into the recruitment mechanism (\b
 *              fibre) and only the entity is put into the filter as
 *              a new value.
 *
 */
void calculate();

/** Calculation of the next value
 *
 *             Calculation of the next value according to the filter
 *             equations.
 *
 *      @param  in double - The new value.
 *
 */
void calculate(double in);

/** Updating of the neuron.
 *
 *              Updating of the neuron either without recruitment
 *              (nothing changed compared to Neuron) or with
 *              recruitment: The calculated value is put as the
 *              normalized one into the learning pathway (\b
 *              normalized \b output) and the output weighted with the
 *              fibres is set as 'real' output. \n
 *              Because of the feedback the number of used fibres may
 *              have changed.
 *
 */
void update();

/** Reseting of the neuron.
 *
 *             Reseting of the neuron.
 *
 */
void reset();

/** Initialization
 *
 *              Initialization of the \b denomiator_ needed for the
 *              calculation of the filter.
 *
 *      @param  f double - The frequency
 *      @param  q double - The quality
 *
 *    @remarks  The quality should be bigger than 0.51
 */
void setFQ(double f, double q);
```

```
    /** Calculation of the normalizing factor.
     *
     *              Calculation of the normalizing factor.
     *
     *    @remarks  The 'search' for the maximum is limited to 200. This
     *              can cause trouble if the frequency is to low.
     */
    void calcNorm();

    // ====================  ACCESS      =========================================


    void setNormalize(bool fnormalize) {normalize_ = fnormalize;};
    bool getNormalize() {return normalize_;};

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! The pre-factor */
    double  denominator_[2];

    /*! The output with history */
    double  buffer_[2];

    /*! The normalizing factor */
    double  norm_;

    /*! A switch for normalizing */
    bool    normalize_;

    static const double  DEF_F = 0.1;
    static const double  DEF_Q = 0.51;

};

#endif
```

## Classes

- class NeuronBP

    *Bandpass Neuron.*

## 6.7 NeuronEligibility.cpp File Reference

```
/** Eligibility Neuron.
 *
 *           \class  NeuronEligibility.cpp
 *
 *                   An eligibility neuron emulates an eligibility
 *                   trace used for Reinforcement learning
 *                   techniques.\n
 *                   If the eligibility value is set to zero only the
 *                   next 4 steps are unequal 0. Otherwise it decays
 *                   exponential with the eligibility value.
 *                   \b eligibility \b value (default) = 0\n
 *
 *            \date  01/07/2007 05:40:17 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "Neuron.h"
#include "Synapse.h"
#include "NeuronEligibility.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// =====================================================================================

/** Calculation of the next output.
 *
 *              Calculation of the \b nextoutput_ using a eligibility
 *              method.
 *
 *
 *    @remarks  If the value of eligibility is 0 the neuron stores a
 *              value only for two step further. Otherwise the buffered value
 *              decays until it reach 0.1.
 */
void NeuronEligibility::calculate()
{

  Neuron::calculate();

  nextoutput_ += lastoutput_;

  // if there is a value unequal 0
  if(eligibility_)
  {

    lastoutput_ = eligibility_ * nextoutput_;
    if(lastoutput_ < 0.01)
    {
      lastoutput_ = 0;
    }

  }
  // if the value is equal 0
```

```
  else
  {
    if(lastoutput_ == 0.5)
    {
      lastoutput_ = 0.25;
    }
    else if(lastoutput_ == 0.25)
    {
      lastoutput_ = 0.1;
    }
    else if(lastoutput_ == 0.1)
    {
      lastoutput_ = 0;
    }
    else
    {
      lastoutput_ = 0.5 * nextoutput_;
    }
  }


}
```

## 6.8 NeuronEligibility.h File Reference

```
/** Eligibility Neuron.
 *
 *        \filename  NeuronEligibility.h
 *
 *           \class  NeuronEligibility.cpp
 *
 *                   An eligibility neuron emulates an eligibility
 *                   trace used for Reinforcement learning
 *                   techniques.\n
 *                   If the eligibility value is set to zero only the
 *                   next 4 steps are unequal 0. Otherwise it decays
 *                   exponential with the eligibility value.
 *                   \b eligibility \b value (default) = 0\n
 *
 *            \date  01/07/2007 05:40:17 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _NEURONELIGIBILITY_H_
#define _NEURONELIGIBILITY_H_

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================

#include "NeuronRecruitment.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class NeuronEligibility : public NeuronRecruitment
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    NeuronEligibility ():eligibility_(0.),lastoutput_(0.){};

    NeuronEligibility(double eligibility):eligibility_(eligibility),lastoutput_(0){};

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    /** Calculation of the next output.
     *
     *              Calculation of the \b nextoutput_ using a eligibility
     *              method.
     *
     *
```

```
     *    @remarks  If the value of eligibility is 0 the neuron stores a
     *              value only for two step further. Otherwise the buffered value
     *              decays until it reach 0.1.
     */
    void calculate();

    // ====================  ACCESS      =========================================

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! The eligiblity factor */
    double eligibility_;

    /*! The buffer of the neuron */
    double lastoutput_;

};

#endif
```

## Classes

- class NeuronEligibility

  *Eligibility Neuron.*

## 6.9 NeuronICO.cpp File Reference

```
/** ICO Neuron.
 *
 *           \class  NeuronICO
 *
 *                   A ICO Neuron that inherit from Neuron.\n
 *                   Learning rule:
 *                   \f[
 *                   \Delta \rho_i(t) = \mu \cdot u_i(t) \cdot u_0^{'}(t)
 *                   \f]
 *                   where \f$\mu\f$ is the learning rate \n
 *                   and \f$\rho_i\f$ is the weight \n
 *                   and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                   predictive inputs\n
 *                   and \f$u_0\f$ is the reflex input.
 *                   \n
 *                   \b learning rate (default) = 1e-4
 *
 *            \date  01/03/2007 08:07:27 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "Neuron.h"
#include "Synapse.h"
#include "NeuronICO.h"


// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// Constructor and Destructor
// =====================================================================================
NeuronICO::NeuronICO()
{

  reset();
  learningRate_ = 1e-4;
  noLearning_ = 0;
  reflex_ = 0;
}

/** Constructor that sets the learning rate.
 *
 *              Constructor that sets the learning rate.
 *
 */
NeuronICO::NeuronICO(double learningRate)
{
  reset();
  learningRate_ = learningRate;
  noLearning_ = 0;
  reflex_ = 0;
}

// =====================================================================================
```

```
// ====================================================================================

/** Calculation of the next output.
 *
 *              Iterates all connected synapses and calculates the \b
 *              nextoutput_ according to the ICO rule.
 *
 *     @return   -
 *
 *    @remarks  The flags are: \n
 *              \b 0 <=> Reflex input (will not be learned) \n
 *              \b 1 <=> Predictive input (will be learned)
 */
void NeuronICO::calculate()
{

  // Iterate all synapses and sum them up into nextoutput_
  nextoutput_ = 0;
  double nextreflex = 0;
  std::list<Synapse*>::iterator i, iend = synapsis_.end();
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
    if (((*i)->getFlags()) == 0)
    {
      nextreflex += (*i)->getOutput();
    }
  }

  if(noLearning_)
    return;

  // Learn
  double derivReflex = nextreflex - reflex_;

  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if (((*i)->getFlags()) == 1)
    {

      (*i)->setWeight((*i)->getWeight() + learningRate_ * derivReflex
            * (*i)->getFactor() * (*i)->getOutput());

    }
  }
  reflex_ = nextreflex;
}
```

## 6.10 NeuronICO.h File Reference

```
/** ICO Neuron.
 *
 *         \filename  NeuronICO.h
 *
 *            \class  NeuronICO
 *
 *                    A ICO Neuron that inherit from Neuron.\n
 *                    Learning rule:
 *                    \f[
 *                    \Delta \rho_i(t) = \mu \cdot u_i(t) \cdot u_0^{'}(t)
 *                    \f]
 *                    where \f$\mu\f$ is the learning rate \n
 *                    and \f$\rho_i\f$ is the weight \n
 *                    and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                    predictive inputs\n
 *                    and \f$u_0\f$ is the reflex input.
 *                    \n
 *                    \b learning rate (default) = 1e-4
 *
 *             \date  01/03/2007 08:07:27 PM CET
 *
 *          \version  1.0
 *           \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                    BCCN Goettingen
 *
 */

#ifndef _NeuronICO_h_
#define _NeuronICO_h_

//----------------------------------------------------------------------
// System Includes
//----------------------------------------------------------------------

#include <iostream>

//----------------------------------------------------------------------
// Project Includes
//----------------------------------------------------------------------

#include "Neuron.h"
//using namespace std

//----------------------------------------------------------------------
// Forward class declarations
//----------------------------------------------------------------------


//----------------------------------------------------------------------
// Implementation
//----------------------------------------------------------------------

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================


// =====================================================================================
// Forward class declarations
// =====================================================================================
```

```
// ======================================================================================
// ======================================================================================
class NeuronICO : public Neuron
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    NeuronICO ();

    NeuronICO(double learningRate);

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    /** Calculation of the next output.
     *
     *              Iterates all connected synapses and calculates the \b
     *              nextoutput_ according to the ICO rule.
     *
     *    @remarks  The flags are: \n
     *              \b 0 <=> Reflex input (will not be learned) \n
     *              \b 1 <=> Predictive input (will be learned)
     */
    virtual void calculate();

    // ====================  ACCESS      =========================================

    void setLearningRate(double learningRate) {learningRate_ = learningRate;};
    double getLearningRate() {return learningRate_;};

    void setNoLearning(bool noLearning) {noLearning_ = noLearning;};
    bool getNoLearning() {return noLearning_;};

    // ====================  INQUIRY     =========================================

  protected:
    /*! The learning rate */
    double learningRate_;

    /*! The reflex input */
    double reflex_;

    /*! A switch for (no) learning */
    bool   noLearning_;

  private:

};

#endif
```

## Classes

- class NeuronICO

  *ICO Neuron.*

## 6.11 NeuronISO.cpp File Reference

```
/** ISO Neuron.
 *
 *           \class  NeuronISO
 *
 *                   A ISO Neuron that inherit from Neuron or ICO
 *                   Neuron, respectively.\n
 *                   Learning rule:
 *                   \f[
 *                   \Delta \rho_i(t) = \mu \cdot u_i(t) \cdot v^{'}(t)
 *                   \f]
 *                   where \f$\mu\f$ is the learning rate \n
 *                   and \f$\rho_i\f$ is the weight \n
 *                   and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                   predictive inputs\n
 *                   and \f$v\f$ is the output.
 *
 *             \see  Neuron
 *             \see  NeuronICO
 *
 *            \date  01/03/2007 07:37:33 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronISO.h"
#include "Synapse.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// =====================================================================================

/** Calculation of the next output.
 *
 *              Iterates all connected synapses and calculates the \b
 *              nextoutput_ according to the ISO rule.
 *
 *     @return   -
 *
 *    @remarks  The flags are: \n
 *              \b 0 <=> Reflex input (will not be learned) \n
 *              \b 1 <=> Predictive input (will be learned)
 */
void NeuronISO::calculate()
{

  // Iterate all synapses and sum them up into nextoutput_
  nextoutput_ = 0;
  std::list<Synapse*>::iterator i, iend = synapsis_.end();
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
  }

  if(noLearning_)
```

```
    return;

  // Learning
  double derivOutput = nextoutput_ - output_;
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if ((*i)->getFlags() == 1)
    {
      (*i)->setWeight((*i)->getWeight() + learningRate_ * derivOutput
            * (*i)->getOutput());

    }
  }
}
```

# 6.12 NeuronISO.h File Reference

```
/** ISO Neuron.
 *
 *        \filename  NeuronISO.h
 *
 *           \class  NeuronISO
 *
 *                   A ISO Neuron that inherit from Neuron or ICO
 *                   Neuron, respectively.\n
 *                   Learning rule:
 *                   \f[
 *                   \Delta \rho_i(t) = \mu \cdot u_i(t) \cdot v^{'}(t)
 *                   \f]
 *                   where \f$\mu\f$ is the learning rate \n
 *                   and \f$\rho_i\f$ is the weight \n
 *                   and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                   predictive inputs\n
 *                   and \f$v\f$ is the output.
 *
 *             \see  Neuron
 *             \see  NeuronICO
 *
 *            \date  01/03/2007 07:37:33 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _NeuronISO_h_
#define _NeuronISO_h_

// =====================================================================================
// System Includes
// =====================================================================================

#include <iostream>

// =====================================================================================
// Project Includes
// =====================================================================================

#include "NeuronICO.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class NeuronISO : public NeuronICO
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    NeuronISO (){};

    NeuronISO(double learningRate):NeuronICO(learningRate){};

    // ====================  OPERATORS   =========================================
```

```
    // ====================  OPERATIONS  =========================================

    /** Calculation of the next output.
     *
     *              Iterates all connected synapses and calculates the \b
     *              nextoutput_ according to the ISO rule.
     *
     *    @remarks  The flags are: \n
     *              \b 0 <=> Reflex input (will not be learned) \n
     *              \b 1 <=> Predictive input (will be learned)
     */
    void calculate();

    // ====================  ACCESS      =========================================

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! The predictive input */
    double preflex_;

};

#endif
```

## Classes

- class NeuronISO

  *ISO Neuron.*

## 6.13 NeuronISO3.cpp File Reference

```
/** ISO-3 Neuron.
 *
 *          \class  NeuronISO3
 *
 *                  A ISO-3 Neuron that inherit from Neuron or ICO
 *                  Neuron, respectively.\n
 *                  Learning rule:
 *                  \f[
 *                  \Delta \rho_i(t) = \mu \cdot r(t) \cdot u_i(t) \cdot v^{'}(t)
 *                  \f]
 *                  where \f$\mu\f$ is the learning rate \n
 *                  and \f$\rho_i\f$ is the weight \n
 *                  and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                  predictive inputs\n
 *                  and \f$v\f$ is the output \n
 *                  and \f$r\f$ is the relevance input.
 *                  \n
 *                  \b learning rate (default) = 1e-4
 *
 *           \date  01/03/2007 08:46:05 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronISO3.h"
#include "Synapse.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// =====================================================================================


/** Calculation of the next output.
 *
 *             Iterates all connected synapses and calculates the \b
 *             nextoutput_ according to the ISO3 rule.
 *
 *     @return   -
 *
 *    @remarks  The flags are: \n
 *             \b 0 <=> Reflex input (will not be learned) \n
 *             \b 1 <=> Predictive input (will be learned) \n
 *             \b 2 <=> Driving input (will not be learned) \n
 *             \b 3 <=> Relevance input (will not be learned and not
 *                      transmitted)
 */
void NeuronISO3::calculate()
{

  // Iterate all synapses and sum them up into nextoutput_
  nextoutput_ = 0;
  std::list<Synapse*>::iterator i, iend = synapsis_.end();
  double R=1;
  double D=1;
```

```
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if ((*i)->getFlags() != 3)
    {
      nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
    }

    if ((*i)->getFlags() == 2)
    {
      D *= (*i)->getWeight() * (*i)->getOutput();
    }

    if ((*i)->getFlags() == 3)
    {
      R *= (*i)->getWeight() * (*i)->getOutput();
      if(R < 0)
        R = 0;

    }

  }

  // No negative current is allowed
  if(nextoutput_ < 0)
  {
    nextoutput_ = 0;
  }

  if(noLearning_)
    return;

  // Learn
  double derivOutput = nextoutput_ - output_;
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if ((*i)->getFlags() == 1)
    {
      (*i)->setWeight((*i)->getWeight() + learningRate_ * derivOutput
            * (*i)->getOutput() * R * D);

    }
  }
}
```

# 6.14 NeuronISO3.h File Reference

```
/** ISO-3 Neuron.
 *
 *        \filename  NeuronISO3.h
 *
 *           \class  NeuronISO3
 *
 *                   A ISO-3 Neuron that inherit from Neuron or ICO
 *                   Neuron, respectively.\n
 *                   Learning rule:
 *                   \f[
 *                   \Delta \rho_i(t) = \mu \cdot r(t) \cdot u_i(t) \cdot v^{'}(t)
 *                   \f]
 *                   where \f$\mu\f$ is the learning rate \n
 *                   and \f$\rho_i\f$ is the weight \n
 *                   and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                   predictive inputs\n
 *                   and \f$v\f$ is the output \n
 *                   and \f$r\f$ is the relevance input.
 *                   \n
 *                   \b learning rate (default) = 1e-4
 *
 *            \date  01/03/2007 08:46:05 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _NeuronISO3_h_
#define _NeuronISO3_h_

//----------------------------------------------------------------------
// System Includes
//----------------------------------------------------------------------

#include <iostream>

//----------------------------------------------------------------------
// Project Includes
//----------------------------------------------------------------------

#include "NeuronICO.h"

//----------------------------------------------------------------------
// Forward class declarations
//----------------------------------------------------------------------


//----------------------------------------------------------------------
// Implementation
//----------------------------------------------------------------------

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================


// =====================================================================================
// Forward class declarations
```

```
// ====================================================================================


// ====================================================================================
// ====================================================================================
class NeuronISO3 : public NeuronICO
{

  public:

    // ====================  LIFECYCLE   =======================================

    /*! Constructor */
    NeuronISO3(){};

    NeuronISO3(double learningRate):NeuronICO(learningRate){};

    // ====================  OPERATORS   =======================================

    // ====================  OPERATIONS  =======================================

    /** Calculation of the next output.
     *
     *              Iterates all connected synapses and calculates the \b
     *              nextoutput_ according to the ISO3 rule.
     *
     *     @return   -
     *
     *    @remarks  The flags are: \n
     *              \b 0 <=> Reflex input (will not be learned) \n
     *              \b 1 <=> Predictive input (will be learned) \n
     *              \b 2 <=> Driving input (will not be learned) \n
     *              \b 3 <=> Relevance input (will not be learned and not
     *                       transmitted)
     */
    void calculate();

    // ====================  ACCESS      =======================================

    // ====================  INQUIRY     =======================================

  protected:

  private:
    double preflex_;

};

#endif
```

## Classes

- class NeuronISO3

    *ISO-3 Neuron.*

# 6.15 NeuronNICO.cpp File Reference

```
/** Normalized ICO Neuron.
 *
 *           \class  NeuronNICO
 *
 *                   A normalized ICO Neuron that inherit from Neuron
 *                   or ICO Neuron, respectively.\n
 *                   Learning rule:
 *                   \f[
 *                   \Delta \rho_i(t) = \mu \cdot N(u_i(t)) \cdot u_0^{'}(t)
 *                   \f]
 *                   where \f$\mu\f$ is the learning rate \n
 *                   and \f$\rho_i\f$ is the weight \n
 *                   and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                   predictive inputs\n
 *                   and \f$N()\f$ is a normalizing function\n
 *                   and \f$u_0\f$ is the reflex input.
 *                   \n
 *                   \b learning rate (default) = 1e-4
 *
 *            \date  01/03/2007 08:28:12 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// ====================================================================================
// Includes
// ====================================================================================

#include "NeuronNICO.h"
#include "Synapse.h"

// ====================================================================================
// defines and consts
// ====================================================================================


// ====================================================================================
// ====================================================================================


/** Calculation of the next output.
 *
 *              Iterates all connected synapses and calculates the \b
 *              nextoutput_ according to the normalized ICO rule.
 *
 *     @return   -
 *
 *    @remarks  The flags are: \n
 *              \b 0 <=> Reflex input (will not be learned) \n
 *              \b 1 <=> Predictive input (will be learned) \n
 *              \b 2 <=> Normalized predictive input (will not be learned)
 */
void NeuronNICO::calculate()
{

  // Iterate all synapses and sum them up into nextoutput_
  nextoutput_ = 0;
  double nextreflex = 0;
  double normalizedpreflex = 0;
  std::list<Synapse*>::iterator i, iend = synapsis_.end();
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
```

```
    nextoutput_ += (*i)->getWeight() * (*i)->getOutput();
    if ((*i)->getFlags() == 0)
    {
      nextreflex += (*i)->getOutput();
    }


    if ((*i)->getFlags() == 2)
    {
      normalizedpreflex = (*i)->getOutput();
    }
  }

  if(noLearning_)
    return;

  // Learn
  double derivReflex = nextreflex - reflex_;

  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if ((*i)->getFlags() == 1)
    {
      (*i)->setWeight((*i)->getWeight() + learningRate_ * derivReflex
            * normalizedpreflex);

    }
  }
  reflex_ = nextreflex;
}
```

## 6.16 NeuronNICO.h File Reference

```
/** Normalized ICO Neuron.
 *
 *        \filename  NeuronNICO.h
 *
 *           \class  NeuronNICO
 *
 *                   A normalized ICO Neuron that inherit from Neuron
 *                   or ICO Neuron, respectively.\n
 *                   Learning rule:
 *                   \f[
 *                   \Delta \rho_i(t) = \mu \cdot N(u_i(t)) \cdot u_0^{'}(t)
 *                   \f]
 *                   where \f$\mu\f$ is the learning rate \n
 *                   and \f$\rho_i\f$ is the weight \n
 *                   and \f$u_i\f$ with \f$i=1..n\f$ are the
 *                   predictive inputs \n
 *                   and \f$N()\f$ is a normalizing function \n
 *                   and \f$u_0\f$ is the reflex input.
 *                   \n
 *                   \b learning rate (default) = 1e-4
 *
 *            \date  01/03/2007 08:28:12 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _NeuronNICO_h_
#define _NeuronNICO_h_

// =====================================================================================
// System Includes
// =====================================================================================

#include <iostream>

// =====================================================================================
// Project Includes
// =====================================================================================

#include "NeuronICO.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class NeuronNICO : public NeuronICO
{

  public:

    // ====================  LIFECYCLE   =======================================

    /*! Constructor */
    NeuronNICO (){};

    NeuronNICO(double learningRate):NeuronICO(learningRate){};

    // ====================  OPERATORS   =======================================
```

```
    // ====================  OPERATIONS  =========================================

    /** Calculation of the next output.
     *
     *              Iterates all connected synapses and calculates the \b
     *              nextoutput_ according to the normalized ICO rule.
     *
     *    @remarks  The flags are: \n
     *              \b 0 <=> Reflex input (will not be learned) \n
     *              \b 1 <=> Predictive input (will be learned) \n
     *              \b 2 <=> Normalized predictive input (will not be learned)
     */
    void calculate();

    // ====================  ACCESS      =========================================

    // ====================  INQUIRY     =========================================

  protected:

  private:

};

#endif
```

## Classes

- class NeuronNICO

  *Normalized ICO Neuron.*

# 6.17 NeuronPID.cpp File Reference

```
/** PID Neuron
 *
 *          \class  NeuronPID.h
 *
 *                  An implementation of an usual PID controller.
 *                  \b set-point (default) = 0 \n
 *                  \b delay (default)   = 0 \n
 *                  \b \f$K_p\f$ (default)   = 400 \n
 *                  \b \f$K_i\f$ (default)   = 2 \n
 *                  \b \f$K_d\f$ (default)   = 0 \n
 *
 *           \date  01/04/2007 02:43:41 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronPID.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// Constructor and Destructor
// =====================================================================================
NeuronPID::NeuronPID()
{
  reset();
  setCoefficients(0, 0, 400, 2, 0);
  setThreshold(0);
}

/** Constructor that sets values.
 *
 *            Constructor that sets values.
 *
 *      @param  setpoint double - The Set point.
 *      @param  timeOffset int - The Delay.
 *      @param  gain double - The gain (Kp)
 *      @param  reset double - The reset (Ki)
 *      @param  deriv double - The derivative (Kd)
 *
 */
NeuronPID::NeuronPID(double setpoint, int timeOffset, double gain, double reset, double deriv)
{
  this->reset();
  setCoefficients(setpoint, timeOffset, gain, reset, deriv);
  setThreshold(0);
}

NeuronPID::~NeuronPID ()
{
}


// =====================================================================================
// =====================================================================================
```

```
/** Calculates the next output.
 *
 *              Calculates the next output. The value provided by the
 *              synapses are put into \b calculate(double)
 *
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronPID::calculate()
{
  Neuron::calculate();
  calculate(nextoutput_);
}


/** Calculation of the next value.
 *
 *              Calculation of the next value given a new input. But
 *              only if the difference between input and set-point is
 *              smaller than \b threshold_
 *
 *      @param  input double - A new value.
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronPID::calculate(double input)
{
  // If deviation is OK
  if(fabs(input - setpoint_) < threshold_)
  {
    input = setpoint_;
  }

  // If delay is bad
  if (timeOffset_ >= MAX_OUTPUT)
  {
    exit(1);
  }

  // Error value shifting
  error_[2] = error_[1];
  error_[1] = error_[0];
  error_[0] = input - setpoint_;

  // PID calculations
  buffer_[timeOffset_] += gain_ * (+error_[0] - error_[1]) + reset_
    * error_[0] + deriv_ * (error_[0] - 2 * error_[1] + error_[2]);

  nextoutput_ = buffer_[0];

  // Buffer shifting
  for (int i = 0; i < timeOffset_; ++i)
  {
    buffer_[i] = buffer_[i + 1];
  }

}

/** Set all values.
 *
 *              Set all values.
 *
 *      @param  setpoint double - The Set point.
 *      @param  timeOffset int - The Delay.
```

```
 *      @param  gain double - The gain (Kp)
 *      @param  reset double - The reset (Ki)
 *      @param  deriv double - The derivative (Kd)
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronPID::setCoefficients(double setpoint, int timeOffset, double gain, double reset, double deriv)
{
  setpoint_ = setpoint;
  timeOffset_ = timeOffset;
  gain_ = gain;
  reset_ = reset;
  deriv_ = deriv;
}


/** Reseting of the neuron.
 *
 *              Reseting of the neuron.
 *
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronPID::reset()
{
  for (int i = 0; i < MAX_OUTPUT; ++i)
  {
    buffer_[i] = 0;
  }

  for (int i = 0; i < 3; ++i)
  {
    error_[i] = 0;
  }
}
```

## 6.18 NeuronPID.h File Reference

```
/** PID Neuron
 *
 *        \filename  NeuronPID.h
 *
 *           \class  NeuronPID.h
 *
 *                   An implementation of an usual PID controller.
 *                   \b set-point (default) = 0 \n
 *                   \b delay (default)   = 0 \n
 *                   \b \f$K_p\f$ (default)   = 400 \n
 *                   \b \f$K_i\f$ (default)   = 2 \n
 *                   \b \f$K_d\f$ (default)   = 0 \n
 *
 *            \date  01/04/2007 02:43:41 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _NeuronPID_h_
#define _NeuronPID_h_

// =========================================================================================
// System Includes
// =========================================================================================

#include <iostream>
#include <cmath>

#define MAX_OUTPUT 10

// =========================================================================================
// Project Includes
// =========================================================================================

#include "NeuronRecruitment.h"

// =========================================================================================
// Forward class declarations
// =========================================================================================


// =========================================================================================
// =========================================================================================
class NeuronPID : public NeuronRecruitment
{

  public:

    // ====================  LIFECYCLE   ========================================

    /*! Constructor */
    NeuronPID ();

    /** Constructor that sets values.
     *
     *              Constructor that sets values.
     *
     *      @param  setpoint double - The Set point.
     *      @param  timeOffset int - The Delay.
     *      @param  gain double - The gain (Kp)
     *      @param  reset double - The reset (Ki)
     *      @param  deriv double - The derivative (Kd)
```

```
     *
     */
    NeuronPID(double setpoint, int timeOffset, double gain,
              double reset, double deriv);

    /*! Destructor */
    ~NeuronPID ();

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    void calculate(double input);
    void calculate();
    void setCoefficients(double setpoint, int timeOffset, double gain,
                         double reset, double deriv);

    void reset();

    // ====================  ACCESS      =========================================

    void setThreshold(double threshold) {threshold_ = threshold;};
    double getThreshold() {return threshold_;};

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*!  */
    double threshold_;

    /*! A buffer for the errors */
    double error_[3];

    /*! A buffer for the output */
    double buffer_[MAX_OUTPUT];

    /*! A set-point */
    double setpoint_;

    /*! A delay factor */
    int    timeOffset_;

    /*! The gain (Kp) */
    double gain_;

    /*! The reset (Ki) */
    double reset_;

    /*! The derivative (Kd) */
    double deriv_;

};

#endif
```

## Classes

- class NeuronPID

    *PID Neuron.*

## 6.19 NeuronQ.cpp File Reference

```
/** Q-Learning Neuron.
 *
 *          \class  NeuronQ
 *
 *                  A Q-Learning Neuron that inherit from Neuron.\n
 *                  Learning rule:
 *                  \f[
 *                  \Delta Q(s,a) = \mu \cdot (Q(s',a') - Q(s,a) + r)
 *                  \f]
 *                  where \f$\mu\f$ is the learning rate \n
 *                  and \f$s\f$ is the current state \n
 *                  and \f$s'\f$ is the next state \n
 *                  and \f$a\f$ is the current action \n
 *                  and \f$a'\f$ is the next action \n
 *                  and \f$r\f$ is the reward\n
 *                  and \f$Q()\f$ are the Q-Values.
 *                  \n
 *                  \b learning rate (default) = 1e-4\n
 *                  Because you have to put the \f$Q(s'a')\f$ into
 *                  the neuron from outside you can decide if it is
 *                  the actual next step (SARSA) or the maximized
 *                  next step (Q-Learning)\n
 *
 *           \date  01/07/2007 01:15:46 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronQ.h"
#include <cmath>

// =====================================================================================
// defines and consts
// =====================================================================================

// =====================================================================================
// Constructor and Destructor
// =====================================================================================
NeuronQ::NeuronQ ()
{
  reset();
  learningRate_ = 1e-4;
  noLearning_ = 0;
}

/** Constructor that sets the learning rate.
 *
 *              Constructor that sets the learning rate.
 *
 */
NeuronQ::NeuronQ(double learningRate)
{
  reset();
  learningRate_ = learningRate;
  noLearning_ = 0;
}

// =====================================================================================
```

```
// ==========================================================================================

/** calculation of the next output.
 *
 *              calculation of the \b nextoutput_ using the sarsa
 *              learning scheme.
 *
 *    @return   -
 *
 *    @remarks  the flags are: \n
 *              \b 0 <=> the value of q(s',a') \n
 *              \b 1 <=> the state variables \n
 *              \b 2 <=> the reward of this state \n
 *              \b 3 <=> this neuron lead to the last action \n
 *              take care of the state values: only the current state
 *              has a value of exact 1. the other values could have
 *              smaller values, though.
 */
void NeuronQ::calculate()
{
  // Iterate all synapses and sum them up into nextoutput_
  nextoutput_ = 0;
  double qValue = 0;
  double action_gate = 0;
  double reward = 0;
  std::list<Synapse*>::iterator i, iend = synapsis_.end();

  // neuron specific
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    // this is the Q-Value of the next step
    if (((*i)->getFlags()) == 0)
    {
      qValue = (*i)->getOutput();
    }

    // reward
    if (((*i)->getFlags()) == 2)
    {
      reward = (*i)->getOutput();
    }

    // this action was conducted
    if (((*i)->getFlags()) == 3)
    {
      action_gate = (*i)->getOutput();
    }

  }

  // synapse specific
  for(i = synapsis_.begin(); i!=iend; ++i)
  {

    if (fabs((*i)->getOutput() - 1) < 1e-10)
    {
      nextoutput_ = (*i)->getWeight();
      (*i)->setFactor(action_gate);
    }
  }



  if(noLearning_)
    return;
```

```
  // Learn
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if (((*i)->getOutput()) < 0.55 && ((*i)->getFlags()) == 1)
    {
      (*i)->setWeight((*i)->getWeight() + learningRate_ * (*i)->getFactor()
          * (*i)->getOutput() * (reward + qValue - (*i)->getWeight()));
    }
  }

}

/** Reseting of the neuron.
 *
 *             Reseting of the neuron.
 *
 *     @return   -
 *
 *    @remarks   -
 */
void NeuronQ::reset()
{
  Neuron::reset();

  std::list<Synapse*>::iterator i, iend = synapsis_.end();
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    (*i)->setFactor(0);
  }


}
```

## 6.20 NeuronQ.h File Reference

```
/** Q-Learning Neuron.
 *
 *       \filename  NeuronQ.h
 *
 *          \class  NeuronQ
 *
 *                  A Q-Learning Neuron that inherit from Neuron.\n
 *                  Learning rule:
 *                  \f[
 *                  \Delta Q(s,a) = \mu \cdot (Q(s',a') - Q(s,a) + r)
 *                  \f]
 *                  where \f$\mu\f$ is the learning rate \n
 *                  and \f$s\f$ is the current state \n
 *                  and \f$s'\f$ is the next state \n
 *                  and \f$a\f$ is the current action \n
 *                  and \f$a'\f$ is the next action \n
 *                  and \f$r\f$ is the reward\n
 *                  and \f$Q()\f$ are the Q-Values.
 *                  \n
 *                  \b learning rate (default) = 1e-4\n
 *                  Because you have to put the \f$Q(s'a')\f$ into
 *                  the neuron from outside you can decide if it is
 *                  the actual next step (SARSA) or the maximized
 *                  next step (Q-Learning)\n
 *
 *           \date  01/07/2007 01:15:46 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

#ifndef _NEURONQ_H_
#define _NEURONQ_H_

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================

#include "Neuron.h"
#include "Synapse.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class NeuronQ : public Neuron
{

  public:

    // ====================  LIFECYCLE   ========================================

    /*! Constructor */
    NeuronQ ();
    NeuronQ(double learningRate);
```

```
    // ====================  OPERATORS   ========================================

    // ====================  OPERATIONS  ========================================

    /** calculation of the next output.
     *
     *              calculation of the \b nextoutput_ using the sarsa
     *              learning scheme.
     *
     *    @remarks  the flags are: \n
     *              \b 0 <=> the value of q(s',a') \n
     *              \b 1 <=> the state variables \n
     *              \b 2 <=> the reward of this state \n
     *              \b 3 <=> this neuron lead to the last action \n
     *              take care of the state values: only the current state
     *              has a value of exact 1. the other values could have
     *              smaller values, though.
     */
    virtual void calculate();


    /** Reseting of the neuron.
     *
     *              Reseting of the neuron.
     *
     */
    void reset();

    // ====================  ACCESS      ========================================

    void setLearningRate(double learningRate) {learningRate_ = learningRate;};
    double getLearningRate() {return learningRate_;};

    void setNoLearning(bool noLearning) {noLearning_ = noLearning;};
    bool getNoLearning() {return noLearning_;};

    // ====================  INQUIRY     ========================================

  protected:

  private:
    /*! The learning rate */
    double learningRate_;

    /*! A switch for (no) learning */
    bool   noLearning_;

};

#endif
```

# Classes

- class NeuronQ

    *Q-Learning Neuron.*

## 6.21 NeuronRecruitment.cpp File Reference

```
/** Recruitment Neuron
 *
 *          \class  NeuronRecruitment
 *
 *                  A recruitment neuron is a special neuron that
 *                  additionally check for a proper feedback value
 *                  which is put into the \b fibres variable.
 *
 *            \see  SynapseRecruitment
 *            \see  Recruitment
 *
 *           \date  01/03/2007 09:16:33 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronRecruitment.h"
#include "Synapse.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// =====================================================================================

/** Set the fibres according to the feedback.
 *
 *             Set the fibres according to the difference of desired
 *             value and current output weighted with the feedback
 *             factor.
 *
 *     @return   -
 *
 *    @remarks  The flags are: \n
 *             \b -1 <=> Feedback input (will not be learned and not
 *                       transmitted)
 */
void NeuronRecruitment::checkFeedback()
{
  std::list<Synapse*>::iterator i, iend = synapsis_.end();
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if((*i)->getFlags() == -1)
    {
      setFibres(getFeedbackFactor() * ((*i)->getOutput() - getDesiredValue()));
    }
  }
}
```

## 6.22 NeuronRecruitment.h File Reference

```
/** Recruitment Neuron
 *
 *       \filename  NeuronRecruitment.h
 *
 *          \class  NeuronRecruitment
 *
 *                  A recruitment neuron is a special neuron that
 *                  additionally check for a proper feedback value
 *                  which is put into the \b fibres variable.
 *
 *            \see  SynapseRecruitment
 *            \see  Recruitment
 *
 *           \date  01/03/2007 09:16:33 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

#ifndef _NeuronRecruitment_h_
#define _NeuronRecruitment_h_

//----------------------------------------------------------------------
// System Includes
//----------------------------------------------------------------------

#include <iostream>

//----------------------------------------------------------------------
// Project Includes
//----------------------------------------------------------------------

#include "Neuron.h"
#include "Recruitment.h"

//----------------------------------------------------------------------
// Forward class declarations
//----------------------------------------------------------------------


//----------------------------------------------------------------------
// Implementation
//----------------------------------------------------------------------

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================


// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class NeuronRecruitment : public Neuron, public Recruitment
{
```

```
  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    NeuronRecruitment (){};

    /*! Destructor */
    virtual ~NeuronRecruitment(){};

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    /** Set the fibres according to the feedback.
     *
     *              Set the fibres according to the difference of desired
     *              value and current output weighted with the feedback
     *              factor.
     *
     *      @return   -
     *
     *     @remarks  The flags are: \n
     *              \b -1 <=> Feedback input (will not be learned and not
     *                        transmitted)
     */
    void checkFeedback();

    // ====================  ACCESS      =========================================

    // ====================  INQUIRY     =========================================

  protected:

  private:

};

#endif
```

# Classes

- class NeuronRecruitment

    *Recruitment Neuron.*

## 6.23 NeuronSigmoid_exp.cpp File Reference

```
/** Sigmoid Neuron (exp)
 *
 *           \class  NeuronSigmoid_exp
 *
 *                   An implementation of a sigmoid neuron between 0
 *                   and 1:
 *                   \f[
 *                   f(x) = \frac{1}{e^{-a (x - b)} - 1}
 *                   \f]
 *                   where \f$a\f$ is a gain (boost)\n
 *                   and \f$b\f$ is a threshold\n
 *
 *                   \b boost (default) = 1 \n
 *                   \b threshold (default)   = 0 \n
 *
 *            \date  01/04/2007 03:47:29 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronRecruitment.h"
#include "Synapse.h"
#include "NeuronSigmoid_exp.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// Constructor and Destructor
// =====================================================================================
NeuronSigmoid_exp::NeuronSigmoid_exp()
{
  reset();
  boost_ = 1;
  threshold_ = 0;
}

/** Constructor that sets values.
 *
 *              Constructor that sets values.
 *
 *      @param  boost double - The gain.
 *      @param  threshold double - The threshold.
 *     @return   -
 *
 *    @remarks   -
 */
NeuronSigmoid_exp::NeuronSigmoid_exp(double boost, double threshold)
{
  reset();
  boost_ = boost;
  threshold_ = threshold;
}


// =====================================================================================
```

```
// ===================================================================================

/** Calculate the next output.
 *
 *              Calculate the \b nextoutput according to the sigmoid
 *              equation.
 *
 *      @return   -
 *
 *     @remarks   -
 */
void NeuronSigmoid_exp::calculate()
{
  Neuron::calculate();

  nextoutput_ = 1/(1 + exp(-boost_ * (nextoutput_ - threshold_)) );

}
```

## 6.24 NeuronSigmoid_exp.h File Reference

```
/** Sigmoid Neuron (exp)
 *
 *       \filename  NeuronSigmoid_exp.h
 *
 *          \class  NeuronSigmoid_exp
 *
 *                  An implementation of a sigmoid neuron between 0
 *                  and 1:
 *                  \f[
 *                  f(x) = \frac{1}{e^{-a (x - b)} - 1}
 *                  \f]
 *                  where \f$a\f$ is a gain (boost)\n
 *                  and \f$b\f$ is a threshold\n
 *
 *                  \b boost (default) = 1 \n
 *                  \b threshold (default)   = 0 \n
 *
 *           \date  01/04/2007 03:47:29 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

#ifndef _NeuronSigmoid_exp_h
#define _NeuronSigmoid_exp_h

//----------------------------------------------------------------------
// System Includes
//----------------------------------------------------------------------

#include <iostream>
#include <cmath>

//----------------------------------------------------------------------
// Project Includes
//----------------------------------------------------------------------

#include "NeuronRecruitment.h"
//using namespace std

//----------------------------------------------------------------------
// Forward class declarations
//----------------------------------------------------------------------


//----------------------------------------------------------------------
// Implementation
//----------------------------------------------------------------------

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================


// =====================================================================================
// Forward class declarations
// =====================================================================================
```

```
// =====================================================================================
// =====================================================================================
class NeuronSigmoid_exp : public NeuronRecruitment
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    NeuronSigmoid_exp ();

    /** Constructor that sets values.
     *
     *              Constructor that sets values.
     *
     *      @param  boost double - The gain.
     *      @param  threshold double - The threshold.
     *
     */
    NeuronSigmoid_exp(double boost, double threshold);

    /*! Destructor */
    ~NeuronSigmoid_exp ();

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    /** Calculate the next output.
     *
     *              Calculate the \b nextoutput according to the sigmoid
     *              equation.
     *
     */
    void calculate();

    // ====================  ACCESS      =========================================

    void setBoost(double fboost) {boost_ = fboost;};
    double getBoost() {return boost_;};
    void setThreshold(double fthreshold) {threshold_ = fthreshold;};
    double getThreshold() {return threshold_;};

    // ====================  INQUIRY     =========================================

  protected:
    /*! The boost (gain) */
    double boost_;

    /*! The threshold */
    double threshold_;

  private:

};

#endif
```

## Classes

- class NeuronSigmoid_exp

    *Sigmoid Neuron (exp).*

## 6.25 NeuronTD.cpp File Reference

```
/** TD-Learning Neuron.
 *
 *          \class  NeuronTD
 *
 *                  A TD-Learning Neuron that inherit from Neuron.\n
 *                  Learning rule:
 *                  \f[
 *                  \Delta V(s) = \mu \cdot (V(s') - V(s) + r)
 *                  \f]
 *                  where \f$\mu\f$ is the learning rate \n
 *                  and \f$s\f$ is the current state \n
 *                  and \f$s'\f$ is the next state \n
 *                  and \f$r\f$ is the reward\n
 *                  and \f$V()\f$ are the V-Values.
 *                  \n
 *                  \b learning rate (default) = 1e-4\n
 *
 *           \date  01/07/2007 01:15:46 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronTD.h"
#include <cmath>

// =====================================================================================
// defines and consts
// =====================================================================================

// =====================================================================================
// Constructor and Destructor
// =====================================================================================
NeuronTD::NeuronTD ()
{
  reset();
  learningRate_ = 1e-4;
  noLearning_ = 0;
}

/** Constructor that sets the learning rate.
 *
 *             Constructor that sets the learning rate.
 *
 */
NeuronTD::NeuronTD(double learningRate)
{
  reset();
  learningRate_ = learningRate;
  noLearning_ = 0;
}

// =====================================================================================
// =====================================================================================

/** calculation of the next output.
 *
 *             calculation of the \b nextoutput_ using the sarsa
 *             learning scheme.
```

```
 *
 *    @return   -
 *
 *    @remarks  the flags are: \n
 *              \b 0 <=> the value of q(s',a') \n
 *              \b 1 <=> the state variables \n
 *              \b 2 <=> the reward of this state \n
 *              \b 3 <=> this neuron lead to the last action \n
 *              take care of the state values: only the current state
 *              has a value of exact 1. the other values could have
 *              smaller values, though.
 */
void NeuronTD::calculate()
{
  // Iterate all synapses and sum them up into nextoutput_
  nextoutput_ = 0;
  double vValue = 0;
  double reward = 0;
  std::list<Synapse*>::iterator i, iend = synapsis_.end();

  // neuron specific
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    // this is the Q-Value of the next step
    if (((*i)->getFlags()) == 0)
    {
      vValue = (*i)->getOutput();
    }

    // reward
    if (((*i)->getFlags()) == 2)
    {
      reward = (*i)->getOutput();
    }

  }

  // synapse specific
  for(i = synapsis_.begin(); i!=iend; ++i)
  {

    if (fabs((*i)->getOutput() - 1) < 1e-10)
    {
      nextoutput_ = (*i)->getWeight();
    }
  }




  if(noLearning_)
    return;

  // Learn
  for(i = synapsis_.begin(); i!=iend; ++i)
  {
    if (((*i)->getOutput()) < 0.55 && ((*i)->getFlags()) == 1)
    {
      (*i)->setWeight((*i)->getWeight() + learningRate_
          * (*i)->getOutput() * (reward + vValue - (*i)->getWeight()));
    }
  }

}
```

# 6.26 NeuronTD.h File Reference

```
/** TD-Learning Neuron.
 *
 *        \filename  NeuronTD.h
 *
 *           \class  NeuronTD
 *
 *                   A TD-Learning Neuron that inherit from Neuron.\n
 *                   Learning rule:
 *                   \f[
 *                   \Delta V(s) = \mu \cdot (V(s') - V(s) + r)
 *                   \f]
 *                   where \f$\mu\f$ is the learning rate \n
 *                   and \f$s\f$ is the current state \n
 *                   and \f$s'\f$ is the next state \n
 *                   and \f$r\f$ is the reward\n
 *                   and \f$V()\f$ are the V-Values.
 *                   \n
 *                   \b learning rate (default) = 1e-4\n
 *
 *            \date  01/07/2007 01:15:46 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _NeuronTD_H_
#define _NeuronTD_H_

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================

#include "Neuron.h"
#include "Synapse.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class NeuronTD : public Neuron
{

  public:

    // ====================  LIFECYCLE   ========================================

    /*! Constructor */
    NeuronTD ();
    NeuronTD(double learningRate);

    // ====================  OPERATORS   ========================================

    // ====================  OPERATIONS  ========================================

    /** calculation of the next output.
```

```
     *
     *              calculation of the \b nextoutput_ using the sarsa
     *              learning scheme.
     *
     *    @remarks  the flags are: \n
     *              \b 0 <=> the value of q(s',a') \n
     *              \b 1 <=> the state variables \n
     *              \b 2 <=> the reward of this state \n
     *              \b 3 <=> this neuron lead to the last action \n
     *              take care of the state values: only the current state
     *              has a value of exact 1. the other values could have
     *              smaller values, though.
     */
    virtual void calculate();


    // ====================  ACCESS      =========================================

    void setLearningRate(double learningRate) {learningRate_ = learningRate;};
    double getLearningRate() {return learningRate_;};

    void setNoLearning(bool noLearning) {noLearning_ = noLearning;};
    bool getNoLearning() {return noLearning_;};

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! The learning rate */
    double learningRate_;

    /*! A switch for (no) learning */
    bool   noLearning_;

};

#endif
```

## Classes

- class NeuronTD

  *TD-Learning Neuron.*

# 6.27 OpenLoop.cpp File Reference

## Functions

- int main (int argc, char *argv[ ])

## 6.27.1 Function Documentation

### 6.27.1.1 int main (int *argc*, char * *argv*[ ])

```
303 {
304
305   //
306   // Initialization of the Neuronal Network and the Recorder
307   //
308
309   // Create a new Neuronal Net!
310   NeuronalNet nnet;
311
312   // Create two bandpass Neuron
313   double reflexF = 0.02;
314   double reflexQ = 0.501;
315   bool   reflexNorm = true;
316   Neuron* nReflex = new NeuronBP(reflexF,reflexQ);
317   ((NeuronBP*)(nReflex))->setNormalize(reflexNorm);
318   nnet.setInput(nReflex);
319
320   double preflexF = 0.01;
321   double preflexQ = 0.501;
322   bool   preflexNorm = true;
323   Neuron* nPreflex = new NeuronBP(preflexF, preflexQ);
324   ((NeuronBP*)(nPreflex))->setNormalize(preflexNorm);
325   nnet.setInput(nPreflex);
326
327
328   // Create two synapses
329   double sReflexWeight = 1;
330   int    sReflexFlags = 0;
331   Synapse* sReflex = new Synapse(nReflex);
332   sReflex->setWeight(sReflexWeight);
333   sReflex->setFlags(sReflexFlags);
334
335   double sPreflexWeight = 0;
336   int    sPreflexFlags = 1;
337   Synapse* sPreflex = new Synapse(nPreflex);
338   sPreflex->setWeight(sPreflexWeight);
339   sPreflex->setFlags(sPreflexFlags);
340
341
342   // Create a learning neuron
343   double learingRate = 1e-1;
344   Neuron* nICO = new NeuronICO(learingRate);
345   nICO->addSynapse(sReflex);
346   nICO->addSynapse(sPreflex);
347   nnet.addNeuron(nICO);
348   nnet.setOutput(nICO);
349
350   // Because we have to input neurons
351   double values[2];
352   double* output;
353
354
355   // Create a Recorder
356   Recorder rec("data.dat");
357
```

```
358     // Add all values of interest out of the Neuronal Net
359     rec.addValue(sPreflex->recWeight(), "Weight of Preflex");
360     rec.addValue(nPreflex, "Output of Preflex");
361     rec.addValue(((NeuronBP*)(nPreflex))->recOutput(), "Output of Synapse");
362     rec.addValue(nICO, "Output");
363 
364     // Create a new recordable container and add  it
365     RecDouble preflex_input;
366     rec.addValue(&preflex_input, "Input to the preflex");
367 
368     // Initialize the recorder
369     rec.init("Weights of the preflex synapse during ICO\n# x0 is shut down at t = 6000");
370 
371 
372     //
373     // Here the simulation starts
374     //
375     int N=10000;
376     srand(1234);
377 
378     for(int i=0; i<N; ++i)
379     {
380 
381 
382       // Every 200 time steps there will be a peak pair
383       // After 6000 the reflexive input vanishes
384       values[0] = (i<6000 ? (i%200==20 ? 1 : 0) : 0);
385       values[1] = (i%200==10 ? 1 : 0);
386 
387 
388       // The container has to be filled
389       preflex_input.setValue(values[1]);
390 
391 
392 
393       // The input values has to be put into the network
394       nnet.setInputValues(values);
395 
396       // then the network has to become updated
397       nnet.update();
398 
399       // the same as the recorder
400       rec.update(i, N, N/10);
401 
402       // afterwards you get the output
403       output = nnet.getOutputValues();
404 
405       // and can use it for further calculations
406       // ..
407       // ..
408 
409     }
410 
411     std::cerr << std::endl;
412 
413 }
```

## 6.28 RecDouble.h File Reference

```
/** Double-valued Recordable.
 *
 *           \class  RecDouble
 *
 *                   A helpfull class that is used to record
 *                   double-type values. \n
 *                   This can be done due to the inheritence of the
 *                   interface Recordable
 *
 *             \see  Recordable
 *             \see  Recorder
 *
 *            \date  01/03/2007 06:33:17 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _RecDouble_h_
#define _RecDouble_h_

//----------------------------------------------------------------------
// System Includes
//----------------------------------------------------------------------

#include <iostream>

//----------------------------------------------------------------------
// Project Includes
//----------------------------------------------------------------------

#include "Recordable.h"

//----------------------------------------------------------------------
// Forward class declarations
//----------------------------------------------------------------------


//----------------------------------------------------------------------
// Implementation
//----------------------------------------------------------------------

// =====================================================================================
// System Includes
// =====================================================================================


// =====================================================================================
// Project Includes
// =====================================================================================


// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class RecDouble : public Recordable
{

  public:
```

```
    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    RecDouble(){};

    RecDouble(double value):value_(value){};

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    // ====================  ACCESS      =========================================
    double record(){return value_;};

    void setValue(double fvalue) {value_ = fvalue; };
    void addValue(double fvalue) {value_ += fvalue;};

    double getValue() {return record();};

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! The value to be recorded */
    double value_;

};

#endif
```

## Classes

- class RecDouble

  *Double-valued Recordable.*

## 6.29 Recordable.h File Reference

```
/** Interface for Recording.
 *
 *           \class  Recordable
 *
 *                   An interface that is used to record values.
 *
 *             \see  Recorder
 *             \see  RecDouble
 *
 *
 *            \date  01/03/2007 06:44:53 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _Recordable_h_
#define _Recordable_h_

// ======================================================================================
// System Includes
// ======================================================================================

#include <iostream>

// ======================================================================================
// Project Includes
// ======================================================================================


// ======================================================================================
// Forward class declarations
// ======================================================================================


// ======================================================================================
// ======================================================================================
class Recordable
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Destructor */
    virtual ~Recordable(){};

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    // ====================  ACCESS      =========================================
    virtual double record() = 0;

    // ====================  INQUIRY     =========================================

  protected:

  private:

};

#endif
```

## Classes

- class Recordable
  *Interface for Recording.*

## 6.30 Recorder.cpp File Reference

```
/** Simple Recorder
 *
 *           \class  Recorder
 *
 *                   A simple recorder that takes Recordables
 *                   (double-typed values) and write them into a
 *                   specified file.
 *
 *             \see  Recordables
 *             \see  RecDouble
 *
 *            \date  01/03/2007 07:04:23 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// ====================================================================================
// Includes
// ====================================================================================

#include "Recorder.h"

// ====================================================================================
// defines and consts
// ====================================================================================


// ====================================================================================
// Constructor and Destructor
// ====================================================================================
Recorder::Recorder(char* filename)
{
  file_.open(filename);
}

Recorder::~Recorder ()
{
  file_.close();
}


// ====================================================================================
// ====================================================================================

/** Write to the file.
 *
 *              Iterate all values and write them into the file.
 *
 *     @return   -
 *
 *    @remarks   -
 */
void Recorder::update()
{
  std::list<Recordable*>::iterator i, iend = values_.end();
  for(i = values_.begin(); i!=iend; ++i)
  {
    file_ << (*i)->record() << "\t";
  }
  file_ << std::endl;
}
```

```
/** Write to the file and show the progress.
 *
 *              Same as \b update() but with a progress indicator.
 *
 *        @see  update()
 *
 *      @param  Nr int - Current position in the loop.
 *      @param  Max int - Number of iterations. (default = 0) \n
 *                        If set, a percentage will be shown. \n
 *                        If not set, the current position will be shown.
 *      @param  interval int - At which interval should the progress
 *                             be shown (default = 1000)
 *     @return   -
 *
 *    @remarks   -
 */
void Recorder::update(int Nr, int Max, int interval)
{
  update();

  if(Nr%interval == interval-1)
  {
    if(Max)
      printf("\r%5.2f%%\t ", Nr*100./((double)(Max)));
    else
      printf("\r%i\t", Nr);

    fflush(NULL);
  }
}

/** Initialize the recording with description.
 *
 *              Initialize the file. Write the descriptions of all
 *              values and an additional description.
 *
 *      @param  mainDescription std::string - An additional
 *                                            description
 *        @see  init()
 *
 *     @return   -
 *
 *    @remarks   -
 */
void Recorder::init(std::string const mainDescription)
{
  file_ << "# " << mainDescription << std::endl;

  file_ << "# ";

  std::list<std::string>::iterator i, iend = descriptions_.end();
  for(i = descriptions_.begin(); i!=iend; ++i)
  {
    file_ << " " << (*i) << " |";
  }
  file_ << std::endl;
}

/** Initialize the recording.
 *
 *              Initialize the file in the same way as \b init() but
 *              without an additional description.
 *
 *        @see  init(std::string)
 *
 *     @return   -
 *
```

```
 *    @remarks   -
 */
void Recorder::init()
{
  init("");
}

/** Add a value to the recorder with a description.
 *
 *              Add a value to the recorder with a description.
 *
 *        @see  addValue(Recordable*)
 *
 *      @param  value Recordable* - A pointer to a Recordable.
 *      @param  description std::string - A description of the value.
 *     @return   -
 *
 *    @remarks   -
 */
void Recorder::addValue(Recordable* value, std::string const description)
{
  values_.push_back(value);
  descriptions_.push_back(description);
}

/** Add a value to the recorder.
 *
 *              Add a value to the recorder in the same way as \b
 *              addValue(Recordable, std::string) but without a
 *              description.
 *
 *        @see  addValue(Recordable*, std::string)
 *
 *      @param  value Recordable* - A pointer to a Recordable.
 *     @return   -
 *
 *    @remarks   -
 */
void Recorder::addValue(Recordable* value)
{
  addValue(value, "");
}
```

## 6.31 Recorder.h File Reference

```
/** Simple Recorder
 *
 *       \filename  Recorder.h
 *
 *          \class  Recorder
 *
 *                  A simple recorder that takes Recordables
 *                  (double-typed values) and write them into a
 *                  specified file.
 *
 *            \see  Recordables
 *            \see  RecDouble
 *
 *           \date  01/03/2007 07:04:23 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

#ifndef _Recorder_h_
#define _Recorder_h_

// =====================================================================================
// System Includes
// =====================================================================================

#include <iostream>
#include <list>
#include <fstream>
#include <string>

// =====================================================================================
// Project Includes
// =====================================================================================

#include "Recordable.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class Recorder
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    Recorder (char* filename);

    /*! Destructor */
    ~Recorder ();

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================


    /** Add a value to the recorder with a description.
```

```
 *
 *              Add a value to the recorder with a description.
 *
 *        @see  addValue(Recordable*)
 *
 *      @param  value Recordable* - A pointer to a Recordable.
 *      @param  description std::string - A description of the value.
 *
 */
void addValue(Recordable* value, std::string const description);

/** Add a value to the recorder.
 *
 *              Add a value to the recorder in the same way as \b
 *              addValue(Recordable, std::string) but without a
 *              description.
 *
 *        @see  addValue(Recordable*, std::string)
 *
 *      @param  value Recordable* - A pointer to a Recordable.
 *
 */
void addValue(Recordable* value);


/** Initialize the recording with description.
 *
 *              Initialize the file. Write the descriptions of all
 *              values and an additional description.
 *
 *      @param  mainDescription std::string - An additional
 *                                            description
 *        @see  init()
 *
 */
void init(std::string const mainDescription);

/** Initialize the recording.
 *
 *              Initialize the file in the same way as \b init() but
 *              without an additional description.
 *
 *        @see  init(std::string)
 *
 */
void init();


/** Write to the file.
 *
 *              Iterate all values and write them into the file.
 *
 */
void update();

/** Write to the file and show the progress.
 *
 *              Same as \b update() but with a progress indicator.
 *
 *        @see  update()
 *
 *      @param  Nr int - Current position in the loop.
 *      @param  Max int - Number of iterations. (default = 0) \n
 *                        If set, a percentage will be shown. \n
 *                        If not set, the current position will be shown.
 *      @param  interval int - At which interval should the progress
 *                             be shown (default = 1000)
```

```
     *
     */
    void update(int Nr, int Max = 0, int interval = 1000);

    // ====================  ACCESS      =========================================

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! A list of values that will be recorded */
    std::list<Recordable*> values_;

    /*! A list of descriptions for each recorded value */
    std::list<std::string> descriptions_;

    /*! The file that will be written */
    std::ofstream file_;

};

#endif
```

## Classes

- class Recorder

  *Simple Recorder.*

## 6.32 Recruitment.h File Reference

```
/** Base Recruitment
 *
 *          \class  Recruitment
 *
 *                  The recruitment mechanism can adopt to different
 *                  external circumstances as long as they behave in
 *                  first order linear. \n
 *                  For this you need a smallest \b entity, a \b maximal
 *                  \b value and a consistent \b number of entities the
 *                  mechanism consist of. \n
 *                  \b entity (default) = 1 \n
 *                  \b max. \b value (default) = 10 \n
 *                  \b nr. \b values (default) = 10 \n
 *                  \b isRecruitment (default) = false \n
 *                  \n
 *                  One possibility to use the recruitment mechanism
 *                  is to predict the upcoming scale of behavior. If
 *                  you 'know' how heavy a cup of water you just put
 *                  this 'knowledge' into the mechanism.\n
 *                  But if your prediction was wrong or you don't
 *                  have any prediction at all you need a feedback
 *                  mechanism that is also provided. A \b desired \b
 *                  value is achieved by activating sufficient \b
 *                  fibres according to the difference of the \b desired
 *                  \b output and the \b current \b output weighted
 *                  by a gain called \b feedback \b factor.\n
 *                  \b desired \b value (default) = 0 \n
 *                  \b feedback \b factor (default) = 1 \n
 *                  \b fibres (default) = 0 \n
 *
 *            \see  NeuronRecruitment
 *            \see  SynapseRecruitment
 *
 *           \date  01/03/2007 08:57:05 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

#ifndef _Recruitment_h_
#define _Recruitment_h_

// =====================================================================================
// System Includes
// =====================================================================================

#include <iostream>

// =====================================================================================
// Project Includes
// =====================================================================================

#include "RecDouble.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================

class RecDouble;


// =====================================================================================
// =====================================================================================
```

```
class Recruitment
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    Recruitment():entity_(1.),maxvalue_(10.),nrValues_(10),isRecruitment_(false),desiredValue_(0),feedback
    {normalizedoutput_.setValue(0.);};

    /*! Destructor */
    ~Recruitment(){};

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    // ====================  ACCESS      =========================================

    void setEntity(double entity) {entity_ = entity;};
    double getEntity() {return entity_;};

    void setMaxvalue(double maxvalue) {maxvalue_ = maxvalue;};
    double getMaxvalue() {return maxvalue_;};

    void setNrValues(int nrValues) {nrValues_ = nrValues;};
    int getNrValues() {return nrValues_;};

    void setNormalizedoutput(double normalizedoutput)
         {normalizedoutput_.setValue(normalizedoutput);};
    double getNormalizedoutput(){return normalizedoutput_.getValue();};

    void setIsRecruitment(bool isRecruitment) {isRecruitment_ = isRecruitment;};
    bool getIsRecruitment() {return isRecruitment_;};

    void setFibres(double fibres) {fibres_ = fibres;};
    double getFibres() {return fibres_;};

    void setDesiredValue(double desiredValue) {desiredValue_ = desiredValue;};
    double getDesiredValue() {return desiredValue_;};

    void setFeedbackFactor(double feedbackFactor)
         {feedbackFactor_ = feedbackFactor;};
    double getFeedbackFactor() {return feedbackFactor_;};

    RecDouble* recOutput(){return &normalizedoutput_;};
    double record(){return getNormalizedoutput();}

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! The basic entity. The smallest value possible. */
    double entity_;

    /*! The biggest value possible */
    double maxvalue_;

    /*! The number of entities needed to reach \b maxvalue_ */
    int    nrValues_;

    /*! A switch for Recruitment */
    bool   isRecruitment_;

    /*! The normalized output */
```

```
    RecDouble normalizedoutput_;

    /*! The desired Value */
    double desiredValue_;

    /*! The feedback factor (gain) */
    double feedbackFactor_;

    /*! The number of fibres needed to reach the \b desiredValue_ with
     * the current output weighted by the \b feedbackFactor_ */
    double fibres_;

};

#endif
```

## Classes

- class Recruitment

  *Base Recruitment.*

## 6.33 Synapse.cpp File Reference

```
/** Basic Synapse
 *
 *          \class  Synapse
 *
 *                  This is the base synapse all other synapse
 *                  inherit from. Basically it has a weight, a flag
 *                  and an output. \n
 *                  \b weight(0) = 1 \n
 *                  \b flag(0)   = 0 \n
 *                  \n
 *                  These properties will not be interpreted by
 *                  the synapse but by the neuron. Thus the output
 *                  from the synapse is not weighted.
 *                  \n
 *                  The synapse itself inherits an interface called
 *                  Recordable. Because of that you can easily record
 *                  the weight of all synapses during learning.
 *
 *            \see  RecDouble
 *            \see  Recordable
 *            \see  Recorder
 *
 *           \date  01/03/2007 05:05:29 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// ======================================================================================
// Includes
// ======================================================================================

#include "Neuron.h"
#include "Synapse.h"

// ======================================================================================
// defines and consts
// ======================================================================================


// ======================================================================================
// Constructor and Destructor
// ======================================================================================
Synapse::Synapse()
{
  setWeight(0.);
  setFlags(0);
  setFactor(1.);
}

/** Constructor that attaches a neuron.
 *
 *             Constructor that attaches a neuron.
 *
 *      @param  neuron Neuron* - A pointer to a neuron.
 *
 */
Synapse::Synapse(Neuron* neuron)
{
  neuron_ = neuron;
  setWeight(0.);
  setFlags(0);
  setFactor(1.);
```

```
}

Synapse::~Synapse ()
{
}


// =====================================================================================
// =====================================================================================



/** Attach the synapse to a neuron.
 *
 *              Attach the synapse to a neuron.
 *
 *      @param  neuron Neuron* - A pointer to a neuron
 *     @return   -
 *
 *    @remarks   -
 */
void Synapse::setNeuron(Neuron* neuron)
{
  neuron_ = neuron;
}
```

# 6.34 Synapse.h File Reference

```
/** Basic Synapse.
 *
 *        \filename  Synapse.h
 *
 *           \class  Synapse
 *
 *                   This is the base synapse all other synapse
 *                   inherit from. Basically it has a weight, a flag
 *                   and an output. \n
 *                   \b weight (default) = 1 \n
 *                   \b flag (default)   = 0 \n
 *                   \n
 *                   These properties will not be interpreted by
 *                   the synapse but by the neuron. Thus the output
 *                   from the synapse is not weighted.
 *                   \n
 *                   The synapse itself inherits an interface called
 *                   Recordable. Because of that you can easily record
 *                   the weight of all synapses during learning.
 *
 *             \see  RecDouble
 *             \see  Recordable
 *             \see  Recorder
 *
 *            \date  01/03/2007 05:05:29 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _Synapse_h_
#define _Synapse_h_

// =====================================================================================
// System Includes
// =====================================================================================

#include <list>
#include <iostream>

// =====================================================================================
// Project Includes
// =====================================================================================

#include "RecDouble.h"
#include "Recordable.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================

class Neuron;

// =====================================================================================
// =====================================================================================
class Synapse : public Recordable
{

  public:

    // ====================  LIFECYCLE   =======================================

    /*! Constructor */
```

```
    Synapse ();

    Synapse(Neuron* neuron);

    /*! Destructor */
    virtual ~Synapse();

    // ====================  OPERATORS   ========================================

    // ====================  OPERATIONS  ========================================

    /** Attach the synapse to a neuron.
     *
     *              Attach the synapse to a neuron.
     *
     *      @param  neuron Neuron* - A pointer to a neuron
     *
     */
    void setNeuron(Neuron* neuron);

    // ====================  ACCESS      ========================================

    virtual double getOutput() {return neuron_->getOutput();};

    void setWeight(double weight) {weight_.setValue(weight);};
    double getWeight() {return weight_.getValue();};

    void setFlags(int flags) {flags_ = flags;};
    int getFlags() {return flags_;};

    void setFactor(double factor) {factor_ = factor;};
    double getFactor() {return factor_;};

    RecDouble* recWeight(){return &weight_;};
    double record(){return getOutput();}

    // ====================  INQUIRY     ========================================

  protected:
    /*! The attached neuron */
    Neuron* neuron_;

    /*! The according weight */
    RecDouble weight_;

    /*! An arbitrary usable flag */
    int flags_;

    /*! A factor that can be used to scale the weight */
    double factor_;

  private:

};

#endif
```

## Classes

- class Synapse

    *Basic Synapse.*

# 6.35 SynapseInput.cpp File Reference

```
/** Sensory (Input) Synapse
 *
 *           \class  SynapseInput
 *
 *                   A sensory synapse that gets it input not from
 *                   other neurons but from the external world.
 *
 *            \date  01/03/2007 06:23:15 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "Neuron.h"
#include "Synapse.h"
#include "SynapseInput.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// Constructor and Destructor
// =====================================================================================
SynapseInput::SynapseInput ()
{
  Synapse::setWeight(1.);
}


// =====================================================================================
// =====================================================================================
```

## 6.36 SynapseInput.h File Reference

```
/** Sensory (Input) Synapse
 *
 *        \filename  SynapseInput.h
 *
 *           \class  SynapseInput
 *
 *                   A sensory synapse that gets it input not from
 *                   other neurons but from the external world.
 *
 *            \date  01/03/2007 06:23:15 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _SynapseInput_h_
#define _SynapseInput_h_

// =====================================================================================
// System Includes
// =====================================================================================

#include <iostream>

// =====================================================================================
// Project Includes
// =====================================================================================

#include "Synapse.h"

// =====================================================================================
// Forward class declarations
// =====================================================================================


// =====================================================================================
// =====================================================================================
class SynapseInput : public Synapse
{

  public:

    // ====================  LIFECYCLE   =========================================

    /*! Constructor */
    SynapseInput ();

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    // ====================  ACCESS      =========================================

    void setOutput(double foutput_) {output_ = foutput_;};
    double getOutput() {return output_;};

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! Since there is no attached neuron the synapse have to have
     * its own input */
```

```
    double output_;

};

#endif
```

# Classes

- class SynapseInput
  *Sensory (Input) Synapse.*

## 6.37 SynapseRecruitment.cpp File Reference

```
/** Recruitment Synapse
 *
 *          \class  SynapseRecruitment
 *
 *                  A recruitment synapse is a special synapse that
 *                  additionally is connected to a recruitment
 *                  neuron.
 *
 *            \see  NeuronRecruitment
 *            \see  Recruitment
 *
 *           \date  01/03/2007 09:07:03 PM CET
 *
 *        \version  1.0
 *         \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                  BCCN Goettingen
 *
 */

// =====================================================================================
// Includes
// =====================================================================================

#include "NeuronRecruitment.h"
#include "SynapseRecruitment.h"

// =====================================================================================
// defines and consts
// =====================================================================================


// =====================================================================================
// Constructor and Destructor
// =====================================================================================
SynapseRecruitment::SynapseRecruitment ()
{
}

/** Constructor that attaches neurons.
 *
 *             Constructor that attaches both a neuron and a
 *             recruitment neuron.
 *
 *     @param  neuronRecruitment NeuronRecruitment* - A pointer to a
 *                                                    recruitment neuron
 *     @param  Neuron* neuron - A pointer to a neuron
 *
 */
SynapseRecruitment::SynapseRecruitment(NeuronRecruitment* neuronRecruitment,
                                       Neuron* neuron)
{
  neuronRecruitment_ = neuronRecruitment;
  neuron_ = neuron;
}

// =====================================================================================
// =====================================================================================
```

## 6.38 SynapseRecruitment.h File Reference

```
/** Recruitment Synapse
 *
 *           \class  SynapseRecruitment
 *
 *                   A recruitment synapse is a special synapse that
 *                   additionally is connected to a recruitment
 *                   neuron.
 *
 *             \see  NeuronRecruitment
 *             \see  Recruitment
 *
 *            \date  01/03/2007 09:07:03 PM CET
 *
 *         \version  1.0
 *          \author  Christoph Kolodziejski (kolo), kolo_at_bccn-goettingen.de
 *                   BCCN Goettingen
 *
 */

#ifndef _SynapseRecruitment_h_
#define _SynapseRecruitment_h_

// ==========================================================================================
// System Includes
// ==========================================================================================

#include <iostream>

// ==========================================================================================
// Project Includes
// ==========================================================================================

#include "Synapse.h"

// ==========================================================================================
// Forward class declarations
// ==========================================================================================

class NeuronRecruitment;


// ==========================================================================================
// ==========================================================================================
class SynapseRecruitment : public Synapse
{

  public:

    // ====================  LIFECYCLE   =======================================

    /*! Constructor */
    SynapseRecruitment ();

    /** Constructor that attaches neurons.
     *
     *               Constructor that attaches both a neuron and a
     *               recruitment neuron.
     *
     *       @param  neuronRecruitment NeuronRecruitment* - A pointer to a
     *                                                     recruitment neuron
     *       @param  neuron Neuron* - A pointer to a neuron
     *
     */
    SynapseRecruitment(NeuronRecruitment* neuronRecruitment, Neuron* neuron);
```

```
    /*! Destructor */
    ~SynapseRecruitment ();

    // ====================  OPERATORS   =========================================

    // ====================  OPERATIONS  =========================================

    // ====================  ACCESS      =========================================

    double getOutput(){return neuronRecruitment_->getNormalizedoutput();};

    // ====================  INQUIRY     =========================================

  protected:

  private:
    /*! An additional recruitment neuron */
    NeuronRecruitment* neuronRecruitment_;

};

#endif
```

## Classes

- class SynapseRecruitment

  *Recruitment Synapse*.

# Index

~Neuron
Neuron, 14
~NeuronPID
NeuronPID, 40
~NeuronRecruitment
NeuronRecruitment, 48
~NeuronSigmoid_exp
NeuronSigmoid_exp, 51
~NeuronalNet
NeuronalNet, 18
~Recordable
Recordable, 59
~Recorder
Recorder, 60
~Recruitment
Recruitment, 65
~Synapse
Synapse, 69
~SynapseRecruitment
SynapseRecruitment, 75

addNeuron
NeuronalNet, 18
addSynapse
Neuron, 14
addValue
RecDouble, 58
Recorder, 61

boost_
NeuronSigmoid_exp, 52
buffer_
NeuronBP, 25
NeuronPID, 42

calcNorm
NeuronBP, 24
calculate
Neuron, 15
NeuronBP, 22
NeuronEligibility, 27
NeuronICO, 30
NeuronISO, 33
NeuronISO3, 35
NeuronNICO, 38
NeuronPID, 40, 41
NeuronQ, 45
NeuronSigmoid_exp, 51
NeuronTD, 54
checkFeedback
NeuronRecruitment, 49
clear
Neuron, 15
NeuronalNet, 20

DEF_F
NeuronBP, 25
DEF_Q
NeuronBP, 25
denominator_
NeuronBP, 25
deriv_
NeuronPID, 43
descriptions_
Recorder, 62
desiredValue_
Recruitment, 67

eligibility_
NeuronEligibility, 27
entity_
Recruitment, 67
error_
NeuronPID, 42

factor_
Synapse, 70
feedbackFactor_
Recruitment, 67
fibres_
Recruitment, 67
file_
Recorder, 63
flags_
Synapse, 70

gain_
NeuronPID, 43
getBoost
NeuronSigmoid_exp, 51
getDesiredValue
Recruitment, 66
getEntity
Recruitment, 65
getFactor
Synapse, 70
getFeedbackFactor
Recruitment, 66
getFibres
Recruitment, 66
getFlags
Synapse, 70
getIsRecruitment
Recruitment, 66
getLearningRate
NeuronICO, 31
NeuronQ, 47
NeuronTD, 55
getMaxvalue
Recruitment, 65
getNoLearning
NeuronICO, 31

NeuronQ, 47
NeuronTD, 55
getNormalize
NeuronBP, 25
getNormalizedoutput
Recruitment, 66
getNrValues
Recruitment, 66
getOutput
Neuron, 16
Synapse, 69
SynapseInput, 72
SynapseRecruitment, 75
getOutputValues
NeuronalNet, 19
getThreshold
NeuronPID, 42
NeuronSigmoid_exp, 52
getValue
RecDouble, 58
getWeight
Synapse, 70

init
Recorder, 61
inputs_
NeuronalNet, 20
isRecruitment_
Recruitment, 67

lastoutput_
NeuronEligibility, 27
learningRate_
NeuronICO, 31
NeuronQ, 47
NeuronTD, 55

main
OpenLoop.cpp, 138
maxvalue_
Recruitment, 67

Neuron, 13
∼Neuron, 14
addSynapse, 14
calculate, 15
clear, 15
getOutput, 16
Neuron, 14
nextoutput_, 16
output_, 16
record, 16
reset, 15
setOutput, 16
synapsis_, 16
update, 15
Neuron.cpp, 77
Neuron.h, 80
neuron_
Synapse, 70
NeuronalNet, 17
NeuronalNet, 18
NeuronalNet
∼NeuronalNet, 18
addNeuron, 18
clear, 20
getOutputValues, 19
inputs_, 20
NeuronalNet, 18
neurons_, 20
outputs_, 20
reset, 20
setInput, 18
setInputValues, 19
setOutput, 18
update, 19
NeuronalNet.cpp, 83
NeuronalNet.h, 87
NeuronBP, 21
NeuronBP, 22
NeuronBP
buffer_, 25
calcNorm, 24
calculate, 22
DEF_F, 25
DEF_Q, 25
denominator_, 25
getNormalize, 25
NeuronBP, 22
norm_, 25
normalize_, 25
reset, 23
setFQ, 23
setNormalize, 25
update, 23
NeuronBP.cpp, 90
NeuronBP.h, 94
NeuronEligibility, 26
NeuronEligibility, 26
NeuronEligibility
calculate, 27
eligibility_, 27
lastoutput_, 27
NeuronEligibility, 26
NeuronEligibility.cpp, 97
NeuronEligibility.h, 99
NeuronICO, 29
NeuronICO, 30
NeuronICO
calculate, 30
getLearningRate, 31
getNoLearning, 31
learningRate_, 31
NeuronICO, 30
noLearning_, 31
reflex_, 31
setLearningRate, 31
setNoLearning, 31
NeuronICO.cpp, 101
NeuronICO.h, 103
NeuronISO, 32
NeuronISO, 33
NeuronISO
calculate, 33
NeuronISO, 33
preflex_, 33
NeuronISO.cpp, 105
NeuronISO.h, 107
NeuronISO3, 34
NeuronISO3, 34, 35
NeuronISO3
calculate, 35
NeuronISO3, 34, 35
preflex_, 36
NeuronISO3.cpp, 109

NeuronISO3.h, 111
NeuronNICO, 37
    NeuronNICO, 37
NeuronNICO
    calculate, 38
    NeuronNICO, 37
NeuronNICO.cpp, 113
NeuronNICO.h, 115
NeuronPID, 39
    NeuronPID, 40
NeuronPID
    ~NeuronPID, 40
    buffer_, 42
    calculate, 40, 41
    deriv_, 43
    error_, 42
    gain_, 43
    getThreshold, 42
    NeuronPID, 40
    reset, 42
    reset_, 43
    setCoefficients, 41
    setpoint_, 43
    setThreshold, 42
    threshold_, 42
    timeOffset_, 43
NeuronPID.cpp, 117
NeuronPID.h, 120
NeuronQ, 44
    NeuronQ, 45
NeuronQ
    calculate, 45
    getLearningRate, 47
    getNoLearning, 47
    learningRate_, 47
    NeuronQ, 45
    noLearning_, 47
    reset, 46
    setLearningRate, 46
    setNoLearning, 47
NeuronQ.cpp, 122
NeuronQ.h, 125
NeuronRecruitment, 48
    NeuronRecruitment, 48
NeuronRecruitment
    ~NeuronRecruitment, 48
    checkFeedback, 49
    NeuronRecruitment, 48
NeuronRecruitment.cpp, 127
NeuronRecruitment.h, 128
neuronRecruitment_
    SynapseRecruitment, 75
neurons_
    NeuronalNet, 20
NeuronSigmoid_exp, 50
    NeuronSigmoid_exp, 51
NeuronSigmoid_exp
    ~NeuronSigmoid_exp, 51
    boost_, 52
    calculate, 51
    getBoost, 51
    getThreshold, 52
    NeuronSigmoid_exp, 51
    setBoost, 51
    setThreshold, 51
    threshold_, 52
NeuronSigmoid_exp.cpp, 130
NeuronSigmoid_exp.h, 132
NeuronTD, 53
    NeuronTD, 54
NeuronTD
    calculate, 54
    getLearningRate, 55
    getNoLearning, 55
    learningRate_, 55
    NeuronTD, 54
    noLearning_, 55
    setLearningRate, 55
    setNoLearning, 55
NeuronTD.cpp, 134
NeuronTD.h, 136
nextoutput_
    Neuron, 16
noLearning_
    NeuronICO, 31
    NeuronQ, 47
    NeuronTD, 55
norm_
    NeuronBP, 25
normalize_
    NeuronBP, 25
normalizedoutput_
    Recruitment, 67
nrValues_
    Recruitment, 67

OpenLoop.cpp, 138
OpenLoop.cpp
    main, 138
output_
    Neuron, 16
    SynapseInput, 73
outputs_
    NeuronalNet, 20

preflex_
    NeuronISO, 33
    NeuronISO3, 36

RecDouble, 57
    RecDouble, 57
RecDouble
    addValue, 58
    getValue, 58
    RecDouble, 57
    record, 58
    setValue, 58
    value_, 58
RecDouble.h, 140
record
    Neuron, 16
    RecDouble, 58
    Recordable, 59
    Recruitment, 67
    Synapse, 70
Recordable, 59
    ~Recordable, 59
    record, 59
Recordable.h, 142
Recorder, 60
    ~Recorder, 60
    addValue, 61
    descriptions_, 62
    file_, 63
    init, 61

Recorder, 60
update, 62
values_, 62
Recorder.cpp, 144
Recorder.h, 147
recOutput
Recruitment, 67
Recruitment, 64
~Recruitment, 65
desiredValue_, 67
entity_, 67
feedbackFactor_, 67
fibres_, 67
getDesiredValue, 66
getEntity, 65
getFeedbackFactor, 66
getFibres, 66
getIsRecruitment, 66
getMaxvalue, 65
getNormalizedoutput, 66
getNrValues, 66
isRecruitment_, 67
maxvalue_, 67
normalizedoutput_, 67
nrValues_, 67
record, 67
recOutput, 67
Recruitment, 65
setDesiredValue, 66
setEntity, 65
setFeedbackFactor, 66
setFibres, 66
setIsRecruitment, 66
setMaxvalue, 65
setNormalizedoutput, 66
setNrValues, 65
Recruitment.h, 150
recWeight
Synapse, 70
reflex_
NeuronICO, 31
reset
Neuron, 15
NeuronalNet, 20
NeuronBP, 23
NeuronPID, 42
NeuronQ, 46
reset_
NeuronPID, 43

setBoost
NeuronSigmoid_exp, 51
setCoefficients
NeuronPID, 41
setDesiredValue
Recruitment, 66
setEntity
Recruitment, 65
setFactor
Synapse, 70
setFeedbackFactor
Recruitment, 66
setFibres
Recruitment, 66
setFlags
Synapse, 70
setFQ
NeuronBP, 23
setInput
NeuronalNet, 18
setInputValues
NeuronalNet, 19
setIsRecruitment
Recruitment, 66
setLearningRate
NeuronICO, 31
NeuronQ, 46
NeuronTD, 55
setMaxvalue
Recruitment, 65
setNeuron
Synapse, 69
setNoLearning
NeuronICO, 31
NeuronQ, 47
NeuronTD, 55
setNormalize
NeuronBP, 25
setNormalizedoutput
Recruitment, 66
setNrValues
Recruitment, 65
setOutput
Neuron, 16
NeuronalNet, 18
SynapseInput, 72
setpoint_
NeuronPID, 43
setThreshold
NeuronPID, 42
NeuronSigmoid_exp, 51
setValue
RecDouble, 58
setWeight
Synapse, 69
Synapse, 68
~Synapse, 69
factor_, 70
flags_, 70
getFactor, 70
getFlags, 70
getOutput, 69
getWeight, 70
neuron_, 70
record, 70
recWeight, 70
setFactor, 70
setFlags, 70
setNeuron, 69
setWeight, 69
Synapse, 69
weight_, 70
Synapse.cpp, 153
Synapse.h, 155
SynapseInput, 72
SynapseInput, 72
SynapseInput
getOutput, 72
output_, 73
setOutput, 72
SynapseInput, 72
SynapseInput.cpp, 157
SynapseInput.h, 158
SynapseRecruitment, 74
SynapseRecruitment, 74

SynapseRecruitment
    ∼SynapseRecruitment, 75
    getOutput, 75
    neuronRecruitment_, 75
    SynapseRecruitment, 74
SynapseRecruitment.cpp, 160
SynapseRecruitment.h, 161
synapsis_
    Neuron, 16

threshold_
    NeuronPID, 42
    NeuronSigmoid_exp, 52
timeOffset_
    NeuronPID, 43

update
    Neuron, 15
    NeuronalNet, 19
    NeuronBP, 23
    Recorder, 62

value_
    RecDouble, 58
values_
    Recorder, 62

weight_
    Synapse, 70