[1. NEURON MODELS 3](#_Toc35872315)

[1.1. NeuronModel 3](#_Toc35872316)

[1.1.1. EventDrivenInputDevice 3](#_Toc35872317)

[1.1.1.1. InputCurrentNeuronModel 3](#_Toc35872318)

[1.1.1.2. InputSpikeNeuronModel 3](#_Toc35872319)

[1.1.2. EventDrivenNeuronModel 3](#_Toc35872320)

[1.1.2.1. CompressTableBasedModel 3](#_Toc35872321)

[1.1.2.1.1. CompressSynchronousTableBasedModel 4](#_Toc35872322)

[1.1.2.2. TableBasedModel 4](#_Toc35872323)

[1.1.2.2.1. SynchronousTableBasedModel 4](#_Toc35872324)

[1.1.3. TimeDrivenModel 4](#_Toc35872325)

[1.1.3.1. TimeDrivenNeuronModel 5](#_Toc35872326)

[1.1.3.1.1. AdExTimeDrivenModel 5](#_Toc35872327)

[1.1.3.1.2. EgidioGranuleCell\_TimeDriven 5](#_Toc35872328)

[1.1.3.1.3. HHTimeDrivenModel 5](#_Toc35872329)

[1.1.3.1.4. IzhikevichTimeDrivenModel 6](#_Toc35872330)

[1.1.3.1.5. LIFTimeDrivenModel 6](#_Toc35872331)

[1.1.3.1.6. LIFTimeDrivenModel\_IS 7](#_Toc35872332)

[1.1.3.1.7. TimeDrivenInferiorOliveCell 7](#_Toc35872333)

[1.1.3.1.8. TimeDrivenPurkinjeCell 8](#_Toc35872334)

[1.1.3.1.9. TimeDrivenPurkinjeCell\_IP 8](#_Toc35872335)

[1.1.3.2. TimeDrivenNeuronModel\_GPU 8](#_Toc35872336)

[1.1.3.2.1. AdExTimeDrivenModel\_GPU 9](#_Toc35872337)

[1.1.3.2.2. EgidioGranuleCell\_TimeDriven\_GPU 9](#_Toc35872338)

[1.1.3.2.3. HHTimeDrivenModel\_GPU 9](#_Toc35872339)

[1.1.3.2.4. IzhikevichTimeDrivenModel\_GPU 10](#_Toc35872340)

[1.1.3.2.5. LIFTimeDrivenModel\_GPU 10](#_Toc35872341)

[1.1.3.2.6. LIFTimeDrivenModel\_IS\_GPU 11](#_Toc35872342)

[1.1.3.2.7. TimeDrivenPurkinjeCell\_GPU 11](#_Toc35872343)

[2. INTEGRATION METHODS 11](#_Toc35872344)

[2.1. IntegrationMethod and IntegrationMethod\_GPU 11](#_Toc35872345)

[2.1.1. IntegrationMethodFast and IntegrationMethodFast\_GPU 12](#_Toc35872346)

[2.1.1.1. ***FixedStep and FixedStep\_GPU*** 12](#_Toc35872347)

[2.1.1.1.1. Euler and Euler\_GPU 12](#_Toc35872348)

[2.1.1.1.2. RK2 and RK2\_GPU 12](#_Toc35872349)

[2.1.1.1.3. RK4 and RK4\_GPU 12](#_Toc35872350)

[2.1.1.1.4. BDF and BDF\_GPU 13](#_Toc35872351)

[2.1.1.2. ***BifixedStep and BifixedStep\_GPU*** 13](#_Toc35872352)

[2.1.1.2.1. Bifixed\_Euler and Bifixed\_Euler\_GPU 13](#_Toc35872353)

[2.1.1.2.2. Bifixed\_RK2 and Bifixed\_RK2\_GPU 13](#_Toc35872354)

[2.1.1.2.3. Bifixed\_RK4 and Bifixed\_RK4\_GPU 14](#_Toc35872355)

[2.1.1.2.4. Bifixed\_BDF 14](#_Toc35872356)

[3. LEARNING RULES 14](#_Toc35872357)

[3.1. LearningRule 14](#_Toc35872358)

[3.1.1. WithoutPostSynaptic 14](#_Toc35872359)

[3.1.1.1. ***AdditiveKernelChange*** 14](#_Toc35872360)

[3.1.1.1.1. ExpWeightChange 14](#_Toc35872361)

[3.1.1.1.2. ExpBufferedWeightChange 14](#_Toc35872362)

[3.1.1.1.3. SinWeightChange and SinBufferedWeightChange 14](#_Toc35872363)

[3.1.1.2. ***CosWeightChange*** 14](#_Toc35872364)

[3.1.1.3. ***SimetricCosWeightChange and SymetricCosBufferedWeightChange*** 14](#_Toc35872365)

[***3.1.1.4.*** ***SimetricCosSinWeightChange*** 14](#_Toc35872366)

[3.1.2. WithPostSynaptic 15](#_Toc35872367)

[3.1.2.1. ***SimetricCosSinSTDPWeightChange*** 15](#_Toc35872368)

[3.1.2.2. ***SimetricCosSTDPWeightChange*** 15](#_Toc35872369)

[3.1.2.3. ***STDPWeightChange*** 15](#_Toc35872370)

[3.1.2.3.1. STDPLSWeightChange 15](#_Toc35872371)

All the variables defined in EDLUT dictionaries are written in lower case using the snake\_case format. Here can be seeing some examples: name, integ\_meth, conf\_filename, tab\_filename, a, b, c, d, e\_exc, e\_inh, c\_m, tau\_exc, tau\_inh, e\_leak, g\_leak, v\_th, tau\_ref, v\_spk\_peak, ep\_cap, thr\_slo\_fac, tau\_w.

# NEURON MODELS

This section defines all the event-driven and time-driven neuron models included in EDLUT with their parameters.

## NeuronModel

This class abstracts the behavior of a neuron in a spiking neural network. It includes internal model functions which define the behavior of the model (initialization, update of the state, synapses effect...). This is only a virtual function (an interface) which defines the functions of the inherited classes.

Parameters:

* newMap["name"] = boost::any(this->name); //string: neuron model name **(AUTOMATICALLY SET BY DERIVED CLASES)**.

### EventDrivenInputDevice

This file declares a class which abstracts an event-driven input device in CPU. These devices can inject input spikes and current in the neural network.

Without parameters.

#### InputCurrentNeuronModel

This class defines the behavior of an input neuron layer that can propagate input currents to the neural network.

Without parameters.

#### InputSpikeNeuronModel

This class defines the behavior of an input neuron layer that can propagate input spikes to the neural network.

Without parameters.

### EventDrivenNeuronModel

This class abstracts the behavior of an event-driven neuron model in a spiking neural network. It includes internal model functions which define the behavior of the model (initialization, update of the state, synapses effect, next firing prediction...). This is only a virtual function (an interface) which defines the functions of the inherited classes.

Without parameters.

#### CompressTableBasedModel

This class implements the behavior of event-driven spiking neuron models using precalculated look-up tables to "predict" the neuron behavior. This "Compress" version can merge several look-up tables with the same indexes in just one, minimizing the look-up time (ideal for complex neuron models such as HH with several state variables and look-up tables).

Parameters:

* newMap["conf\_filename"] = boost::any(this->conf\_filename); //String: name of configuration file.cfg where is defined the look-up tables structure
* newMap["tab\_filename"] = boost::any(this->tab\_filename); //String: name of file.dat where are defined the look-up tables.

#### CompressSynchronousTableBasedModel

This class implements the behavior of event-driven spiking neuron models using precalculated look-up tables to "predict" the neuron behavior. This "CompressSynchronous" version not only can merge several look-up tables with the same indexes in just one, such as his parent class, but also can synchronize the output activity, minimizing the time required to process the output spikes.

Parameters:

* newMap["conf\_filename"] = boost::any(this->conf\_filename); //String: name of configuration file.cfg where is defined the look-up tables structure
* newMap["tab\_filename"] = boost::any(this->tab\_filename); //String: name of file.dat where are defined the look-up tables.

#### TableBasedModel

This class implements the behavior of event-driven spiking neuron models using precalculated look-up tables to "predict" the neuron behavior.

Parameters:

* newMap["conf\_filename"] = boost::any(this->conf\_filename); //String: name of configuration file.cfg where is defined the look-up tables structure
* newMap["tab\_filename"] = boost::any(this->tab\_filename); //String: name of file.dat where are defined the look-up tables.

#### SynchronousTableBasedModel

This class implements the behavior of event-driven spiking neuron models using precalculated look-up tables to "predict" the neuron behavior. This "Synchronous" version can synchronize the output activity, minimizing the time required to process the output spikes.

Parameters:

* newMap["conf\_filename"] = boost::any(this->conf\_filename); //String: name of configuration file.cfg where is defined the look-up tables structure
* newMap["tab\_filename"] = boost::any(this->tab\_filename); //String: name of file.dat where are defined the look-up tables.

### TimeDrivenModel

This class abstracts the behavior of time-driven neuron models in spiking neural networks. It includes internal model functions which define the behavior of the model (initialization, update of the state, synapses effect...). This is only a virtual function (an interface) which defines the functions of the inherited classes.

Without parameters.

#### TimeDrivenNeuronModel

This class abstracts the behavior of time-driven neuron models in spiking neural networks implemented in CPU. It includes internal model functions which define the behavior of the model (initialization, update of the state, synapses effect...). This is only a virtual function (an interface) which defines the functions of the inherited classes.

Parameters:

* newMap["int\_meth"] = imethod; //INTEGRATION METHODS DEFINED IN SECTION 2.

##### AdExTimeDrivenModel

This class implement an AdEx Time-Driven neuron model with a membrane potential (V), a membrane recovery variable (w), three conductances (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["a"] = boost::any(this->a); //conductance (nS)
* newMap["b"] = boost::any(this->b); //spike trigger adaptation (pA)
* newMap["thr\_slo\_fac"] = boost::any(this->thr\_slo\_fac); //threshold slope factor (mV)
* newMap["v\_thr"] = boost::any(this->v\_thr); //effective threshold potential (mV)
* newMap["tau\_w"] = boost::any(this->tau\_w); //adaptation time constant (ms)
* newMap["e\_exc"] = boost::any(this->e\_exc); //excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); //inhibitory reversal potential (mV)
* newMap["e\_reset"] = boost::any(this->e\_reset); //reset potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); //effective leak potential (mV)
* newMap["g\_leak"] = boost::any(this->g\_leak); //leak conductance (nS)
* newMap["c\_m"] = boost::any(this->c\_m); //membrane capacitance (pF)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); //AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); //GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); //NMDA (excitatory) receptor time constant (ms)

##### EgidioGranuleCell\_TimeDriven

This class implements a detailed Leaky Integrate-And-Fire Time-Driven neuron model for a cerebellar granule cell. This neuron model includes 15 state variables modelling the neural dynamics, three conductances (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Without parameters.

##### HHTimeDrivenModel

This class implements a Hodgkin and Huxley neuron model with four differential equations (membrane potential, m, h and n), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); //Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential(mV)
* newMap["g\_leak"] = boost::any(this->g\_leak); // Leak conductance (nS)
* newMap["c\_m"] = boost::any(this->c\_m); // Membrane capacitance (pF)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory NMDA) receptor time constant (ms)
* newMap["g\_na"] = boost::any(this->g\_na); // Maximum value of sodium conductance (nS)
* newMap["g\_kd"] = boost::any(this->g\_kd); // Maximum value of potassium conductance (nS)
* newMap["e\_na"] = boost::any(this->e\_na); // Sodium potential (mV)
* newMap["e\_k"] = boost::any(this->e\_k); // Potassium potential (mV)

##### IzhikevichTimeDrivenModel

This class implements an Izhikevich neuron model with two differential equations (membrane potential and membrane recovery), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["a"] = boost::any(this->a); // Time scale of recovery variable u (dimensionless)
* newMap["b"] = boost::any(this->b); // Sensitivity of the recovery variable u to the subthreshold fluctuations of the membrane potential v (dimensionless)
* newMap["c"] = boost::any(this->c); // After-spike reset value of the membrane potential v (dimensionless)
* newMap["d"] = boost::any(this->d); // After-spike reset of the recovery variable u (dimensionless)
* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["c\_m"] = boost::any(this->c\_m); // Membrane capacitance (pF)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)

##### LIFTimeDrivenModel

This class implements a Leaky Integrate-And-Fire (LIF) neuron model with one differential equation (membrane potential), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (mV)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["c\_m"] = boost::any(float(this->c\_m)); // Membrane capacitance (pF)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (ms)
* newMap["g\_leak"] = boost::any(float(this->g\_leak)); // Leak conductance (nS)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)

##### LIFTimeDrivenModel\_IS

This class implements a Leaky Integrate-And-Fire (LIF) neuron model with one differential equation (membrane potential), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse. In this case, the model uses the units defined in the International System (IS): seconds (s), volts (V), faradiums (F) and siemens (S) instead of miliseconds (ms), milivolts (mV), picofaradiums (pF) and nanosiemens (nS).

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (V)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (V)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (V)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (V)
* newMap["c\_m"] = boost::any(float(this->c\_m)); // Membrane capacitance (F)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (s)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (s)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (s)
* newMap["g\_leak"] = boost::any(float(this->g\_leak)); // Leak conductance (S)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (s)

##### TimeDrivenInferiorOliveCell

This class implements an Inferior Olive cell as a Leaky Integrate-And-Fire (LIF) neuron model with one differential equation (membrane potential), three time dependent equations (excitatory, inhibitory and NMDA conductances), one external input current synapse, and one electrical coupling synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (mV)
* newMap["c\_m"] = boost::any(this->c\_m); // Membrane capacitance (uF/cm^2)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (ms)
* newMap["g\_leak"] = boost::any(this->g\_leak); // Leak conductance (mS/cm^2)
* newMap["area"] = boost::any(this->area); // Cell area (cm^2)

##### TimeDrivenPurkinjeCell

This class implements a Purkinje cell model with three differential equations (membrane potential, calcium and Muscariny channels), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (mV)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (ms)

##### TimeDrivenPurkinjeCell\_IP

This class implements a Purkinje cell model with four differential equations (membrane potential, calcium and Muscariny channels and membrane capacitance), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (mV)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (ms)
* newMap["ep\_c\_m"] = boost::any(this->ep\_c\_m); // Epsilon capacitance (uF\*ms/cm^2)

#### TimeDrivenNeuronModel\_GPU

This class abstracts the behavior of time-driven neuron models in spiking neural networks implemented in GPU (with an additional C\_INTERFACE class in CPU to manage the GPU class). It includes internal model functions which define the behavior of the model (initialization, update of the state, synapses effect...). This is only a virtual function (an interface) which defines the functions of the inherited classes.

Parameters:

* newMap["int\_meth"] = imethod; //INTEGRATION METHODS DEFINED IN SECTION 2.

##### AdExTimeDrivenModel\_GPU

This class implement an AdEx Time-Driven neuron model with a membrane potential (V), a membrane recovery variable (w), three conductances (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["a"] = boost::any(this->a); //conductance (nS)
* newMap["b"] = boost::any(this->b); //spike trigger adaptation (pA)
* newMap["thr\_slo\_fac"] = boost::any(this->thr\_slo\_fac); //threshold slope factor (mV)
* newMap["v\_thr"] = boost::any(this->v\_thr); //effective threshold potential (mV)
* newMap["tau\_w"] = boost::any(this->tau\_w); //adaptation time constant (ms)
* newMap["e\_exc"] = boost::any(this->e\_exc); //excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); //inhibitory reversal potential (mV)
* newMap["e\_reset"] = boost::any(this->e\_reset); //reset potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); //effective leak potential (mV)
* newMap["g\_leak"] = boost::any(this->g\_leak); //leak conductance (nS)
* newMap["c\_m"] = boost::any(this->c\_m); //membrane capacitance (pF)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); //AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); //GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); //NMDA (excitatory) receptor time constant (ms)

##### EgidioGranuleCell\_TimeDriven\_GPU

This class implements a detailed Leaky Integrate-And-Fire Time-Driven neuron model for a cerebellar granule cell. This neuron model includes 15 state variables modelling the neural dynamics, three conductances (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Without parameters.

##### HHTimeDrivenModel\_GPU

This class implements a Hodgkin and Huxley neuron model with four differential equations (membrane potential, m, h and n), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); //Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential(mV)
* newMap["g\_leak"] = boost::any(this->g\_leak); // Leak conductance (nS)
* newMap["c\_m"] = boost::any(this->c\_m); // Membrane capacitance (pF)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory NMDA) receptor time constant (ms)
* newMap["g\_na"] = boost::any(this->g\_na); // Maximum value of sodium conductance (nS)
* newMap["g\_kd"] = boost::any(this->g\_kd); // Maximum value of potassium conductance (nS)
* newMap["e\_na"] = boost::any(this->e\_na); // Sodium potential (mV)
* newMap["e\_k"] = boost::any(this->e\_k); // Potassium potential (mV)

##### IzhikevichTimeDrivenModel\_GPU

This class implements an Izhikevich neuron model with two differential equations (membrane potential and membrane recovery), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["a"] = boost::any(this->a); // Time scale of recovery variable u (dimensionless)
* newMap["b"] = boost::any(this->b); // Sensitivity of the recovery variable u to the subthreshold fluctuations of the membrane potential v (dimensionless)
* newMap["c"] = boost::any(this->c); // After-spike reset value of the membrane potential v (dimensionless)
* newMap["d"] = boost::any(this->d); // After-spike reset of the recovery variable u (dimensionless)
* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["c\_m"] = boost::any(this->c\_m); // Membrane capacitance (pF)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)

##### LIFTimeDrivenModel\_GPU

This class implements a Leaky Integrate-And-Fire (LIF) neuron model with one differential equation (membrane potential), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (mV)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["c\_m"] = boost::any(float(this->c\_m)); // Membrane capacitance (pF)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (ms)
* newMap["g\_leak"] = boost::any(float(this->g\_leak)); // Leak conductance (nS)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)

##### LIFTimeDrivenModel\_IS\_GPU

This class implements a Leaky Integrate-And-Fire (LIF) neuron model with one differential equation (membrane potential), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse. In this case, the model uses the units defined in the International System (IS): seconds (s), volts (V), faradiums (F) and siemens (S) instead of miliseconds (ms), milivolts (mV), picofaradiums (pF) and nanosiemens (nS).

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (V)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (V)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (V)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (V)
* newMap["c\_m"] = boost::any(float(this->c\_m)); // Membrane capacitance (F)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (s)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (s)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (s)
* newMap["g\_leak"] = boost::any(float(this->g\_leak)); // Leak conductance (S)

newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (s)

##### TimeDrivenPurkinjeCell\_GPU

This class implements a Purkinje cell model with three differential equations (membrane potential, calcium and Muscariny channels), three time dependent equations (excitatory, inhibitory and NMDA conductances) and one external input current synapse.

Parameters:

* newMap["e\_exc"] = boost::any(this->e\_exc); // Excitatory reversal potential (mV)
* newMap["e\_inh"] = boost::any(this->e\_inh); // Inhibitory reversal potential (mV)
* newMap["v\_thr"] = boost::any(this->v\_thr); // Effective threshold potential (mV)
* newMap["e\_leak"] = boost::any(this->e\_leak); // Effective leak potential (mV)
* newMap["tau\_exc"] = boost::any(this->tau\_exc); // AMPA (excitatory) receptor time constant (ms)
* newMap["tau\_inh"] = boost::any(this->tau\_inh); // GABA (inhibitory) receptor time constant (ms)
* newMap["tau\_nmda"] = boost::any(this->tau\_nmda); // NMDA (excitatory) receptor time constant (ms)
* newMap["tau\_ref"] = boost::any(this->tau\_ref); // Refractory period (ms)

# INTEGRATION METHODS

This section defines the integration methods used in all the time-driven neuron models, both in CPU and GPU. These methods are classified in two sub-groups: fixed-step and bifixed-step integration methods.

* 1. **IntegrationMethod and IntegrationMethod\_GPU**

These classes respectively abstract the behavior of all the integration methods in CPU and GPU for all the time-driven neural models defined in CPU and GPU. They include internal model functions which define the behavior of the integration methods (initialization, calculate next value, ...). These are only virtual functions (interfaces) which define the functions of the inherited classes.

Parameters:

* newMap["step"] = boost::any(this->elapsedTimeInSeconds); // Fixed-step size defined in seconds (larger step size in bifixed methods)
* newMap["name"] = boost::any(this->name); // Integration method name **(AUTOMATICALLY SET BY DERIVED CLASES)**.

### IntegrationMethodFast and IntegrationMethodFast\_GPU

These intermediate classes are respectively defined in CPU and GPU to include a template pointer to the neuron models that must be integrated. This configuration using templates allows the CPU and GPU compilers to perform several optimizations in the integration process for each specific combination of neuron model and integration methods (loops unrolling, inline functions even with inherit virtual methods, etc.). All the possible neuron model and integration method combinations are set in the IntegrationMethodFactory and IntegrationMethodFactory\_GPU classes.

Without parameters.

* + - 1. ***FixedStep and FixedStep\_GPU***

These classes abstract the behavior of all the fixed-step integration methods for time-driven neural models both in CPU and GPU. They include internal model functions which define the behavior of integration methods (initialization, calculate next value, ...). These are only virtual classes (interfaces) which define the functions of the inherited classes.

Without Parameters.

##### Euler and Euler\_GPU

These classes implement a first order Euler integration method able to integrate the differential equations that define the time-driven neuron models both in CPU and GPU. This method is the most computational efficient, but also the lesser precise.

The name of these classes is **“Euler”** for both implementations.

Without parameters.

##### RK2 and RK2\_GPU

These classes implement a second order Runge-Kutta integration method able to integrate the differential equations that define the time-driven neuron models both in CPU and GPU. This method is more complex and precise than the Euler method.

The name of these classes is **“RK2”** for both implementations.

Without parameters.

##### RK4 and RK4\_GPU

These classes implement a fourth order Runge-Kutta integration method able to integrate the differential equations that define the time-driven neuron models both in CPU and GPU. This method is more complex and precise than the RK2 method.

The name of these classes is **“RK4”** for both implementations.

Without parameters.

##### BDF and BDF\_GPU

These classes implement six implicit Backward Differentiation Formula integration methods (from first to sixth order) able to integrate the differential equations that define the time-driven neuron models both in CPU and GPU. These implicit methods are much more complex and precise that the previous explicated methods and are oriented to compute very complex neuron models with stiff differential equations (HH and more complex models).

The name of these classes is **“DBF”** for both implementations.

Parameters:

* newMap["bdf\_order"] = this->BDForder; // Bdf order (from 1 to 6)
  + - 1. ***BifixedStep and BifixedStep\_GPU***

These classes abstract the behavior of all the bifixed-step integration methods for time-driven neural models both in CPU and GPU. They include internal model functions which define the behavior of integration methods (initialization, calculate next value, ...). These are only virtual classes (interfaces) which define the functions of the inherited classes.

These integration methods are defines as bifixed-step because they use two different fixed-step values. The larger step is the “step” defined in the “IntegrationMethod” and “IntegrationMethod\_GPU” classes. By contrast, the smaller step is a divisor of the larger step and is computed with the parameter “n\_steps” that sets in how many sections must be divided the larger step to obtain the small one. Thus, each integration step can be done with just one larger step of with n\_steps smaller steps. The implementation in CPU and GPU is a bit different. For more information, see <https://www.frontiersin.org/articles/10.3389/fninf.2017.00007/full>

Parameters:

* newMap["n\_steps"] = boost::any(this->ratioLargerSmallerSteps); // Ration between the larger and the smaller integration steps.

##### Bifixed\_Euler and Bifixed\_Euler\_GPU

These classes implement a first order Euler integration method able to integrate the differential equations that define the time-driven neuron models both in CPU and GPU. This method is the most computational efficient, but also the lesser precise.

The name of these classes is **“Bifixed\_Euler”** for both implementations.

Without parameters.

##### Bifixed\_RK2 and Bifixed\_RK2\_GPU

These classes implement a second order Runge-Kutta integration method able to integrate the differential equations that define the time-driven neuron models both in CPU and GPU. This method is more complex and precise than the Euler method.

The name of these classes is **“Bifixed\_RK2”** for both implementations.

Without parameters.

##### Bifixed\_RK4 and Bifixed\_RK4\_GPU

These classes implement a fourth order Runge-Kutta integration method able to integrate the differential equations that define the time-driven neuron models both in CPU and GPU. This method is more complex and precise than the RK2 method.

The name of these classes is **“Bifixed\_RK4”** for both implementations.

Without parameters.

##### Bifixed\_BDF

These classes implement two implicit Backward Differentiation Formula integration methods (first and second order) able to integrate the differential equations that define the time-driven neuron models **just in CPU**. These implicit methods are much more complex and precise that the previous explicated methods and are oriented to compute very complex neuron models with stiff differential equations (HH and more complex models).

The name of these classes is **“Bifixed\_DBF”** for both implementations.

Parameters:

* newMap["bdf\_order"] = this->BDForder; // Bdf order (from 1 to 2)

# LEARNING RULES

A learning rule is a plasticity mechanism that adjusts the synaptic strength (synaptic weights) in between spiking neurons using different spike-timing-dependent mechanisms.

A learning rule in spiking neural networks is a biological process that adjusts the strength of connections between neurons. This process adjusts the connection weight based on the relative timing of two events in a particular neuron (depending on the learning mechanism). These learning rules can be classified in two main groups: supervised and unsupervised learning rules.

* The supervised learning rules correlate the arrival times of input spikes coming from **normal** and **trigger** connections. Only normal connections modify their weights.
* The unsupervised learning rules correlate the arrival times of input spikes coming from **normal** connections and the generation time of output spikes in the target neuron.

This learning process partially explains the activity-dependent development of nervous systems, especially with regards to long-term potentiation (LTP) and long-term depression (LTD) effects experimentally observed.

## LearningRule

This class abstracts the behavior of a learning rule in a spiking neural network. It includes internal model functions which define the behavior of the model (initialization, update of the state, etc.). This is only a virtual class (an interface) which defines the functions of the inherited classes.

Without parameters.

### WithoutPostSynaptic

This file declares a class which abstracts a learning rule without postsynaptic learning. In this case, each learning rule has two types of input synapses: normal and trigger synapses. When a spike reaches a target neuron through a normal synapse that implement a learning rule of this type, this synapse updates its weight considering this activity and the learning rule parameters (LTP or LTD). By contrast, when the spike is propagated by a trigger connection toward a target neuron, this spike throws another learning mechanism (LTP or LTD) over all the normal input synapses associated to this learning rule considering their past (and in some cases also their future) presynaptic activity.

Normal connections are indicated in the network definition using the learning rule index. Trigger connections are indicated in the network definition using a "t" + the learning rule index.

Without parameters.

* + - 1. ***AdditiveKernelChange***

Parameters:

* newMap["kernel\_peak"] = boost::any(this->kernelpeak);
* newMap["fixed\_change"] = boost::any(this->fixwchange);
* newMap["kernel\_change"] = boost::any(this->kernelwchange);

##### ExpWeightChange

Without parameters:

##### ExpBufferedWeightChange

Parameters:

* newMap["init\_time"] = boost::any(this->initTime);

##### SinWeightChange and SinBufferedWeightChange

* newMap["exp"] = boost::any(this->exponent);
  + - 1. ***CosWeightChange***

Parameters:

* newMap["tau"] = boost::any(this->tau);
* newMap["exp"] = boost::any(this->exponent);
* newMap["fixed\_change"] = boost::any(this->fixwchange);
* newMap["kernel\_change"] = boost::any(this->kernelwchange);
  + - 1. ***SimetricCosWeightChange and SymetricCosBufferedWeightChange***

Parameters:

* newMap["tau"] = boost::any(this->tau);
* newMap["exp"] = boost::any(this->exponent);
* newMap["fixed\_change"] = boost::any(this->fixwchange);
* newMap["kernel\_change"] = boost::any(this->kernelwchange);
  + - 1. ***SimetricCosSinWeightChange***

Parameters:

* newMap["max\_min\_dist"] = boost::any(this->MaxMinDistance);
* newMap["central\_amp"] = boost::any(this->CentralAmplitudeFactor);
* newMap["lateral\_amp"] = boost::any(this->LateralAmplitudeFactor);

### WithPostSynaptic

This class abstracts a learning rule that implements postsynaptic learning. In this case, each learning rule has just one type of input synapses: normal synapses. When a spike reaches a target neuron through a normal presynaptic connection that implement a learning rule of this type, this connection correlates the spike time with the previous postsynaptic output spike times, thus generating the corresponding LTP or LTD response in function of the learning rule kernel shape. Additionally, when the target neuron generates a postsynaptic spike, this neuron checks the presynaptic activity of all its normal input connections and generates the corresponding LTP or LTD response in each synapse.

Without parameters.

* + - 1. ***SimetricCosSinSTDPWeightChange***

Parameters:

* newMap["max\_min\_dist"] = boost::any(this->MaxMinDistance);
* newMap["central\_amp"] = boost::any(this->CentralAmplitudeFactor);
* newMap["lateral\_amp"] = boost::any(this->LateralAmplitudeFactor);
  + - 1. ***SimetricCosSTDPWeightChange***

Parameters:

* newMap["tau"] = boost::any(this->tau);
* newMap["exp"] = boost::any(this->exponent);
* newMap["fixed\_change"] = boost::any(this->fixwchange);
* newMap["kernel\_change"] = boost::any(this->kernelwchange);
  + - 1. ***STDPWeightChange***

Parameters:

* newMap["max\_LTP"] = boost::any(this->MaxChangeLTP);
* newMap["tau\_LTP"] = boost::any(this->tauLTP);
* newMap["max\_LTD"] = boost::any(this->MaxChangeLTD);
* newMap["tau\_LTD"] = boost::any(this->tauLTD);

##### STDPLSWeightChange

Parameters:

* newMap["max\_LTP"] = boost::any(this->MaxChangeLTP);
* newMap["tau\_LTP"] = boost::any(this->tauLTP);
* newMap["max\_LTD"] = boost::any(this->MaxChangeLTD);
* newMap["tau\_LTD"] = boost::any(this->tauLTD);

# NETWORK DEFINITION

To define a new layer on EDLUT, the user must call the nest functions “AddSynapticLayer(source\_neurons, target\_neurons, synaptic\_params)” where source\_neurons and target\_neurons are two vectors of the same size with the global neuron indexes of the source and target neurons. Regarding the synaptic\_params parameters, it is a dictionary with the next parameters:

* synaptic\_layer.param\_map["delay"] = boost::any(delay\_list); //DEFAULT = 0.001 synaptic\_layer.param\_map["type"] = boost::any(type\_list); //DEFAULT = 0
* synaptic\_layer.param\_map["weight"] = boost::any(weight\_list); //DEFAULT = 1.0
* synaptic\_layer.param\_map["max\_weight"] = boost::any(max\_weight\_list); //DEFAULT = 1.0
* synaptic\_layer.param\_map["trigger\_wchange"] = boost::any(triggerwchange\_list); //DEFAULT = -1
* synaptic\_layer.param\_map["wchange"] = boost::any(wchange\_list); //DEFAULT = -1

The values assigned to each parameter can independently be a single value or a vector of values with the same size that the source\_neurons and targer\_neruons vectors. Finally, if some value it is not assigned, this one takes the DEFAULT value.