Neuron and synapse models in NESTML: From specification to simulation

Charl Linssen <c.linssen@fz-juelich.de> | EBRAINS User Day, Heidelberg | March 12th, 2025

JÜLICH
Forschungszentrum

Image credit: (c) S. Spreizer

Modeling and simulation with NESTML

What?

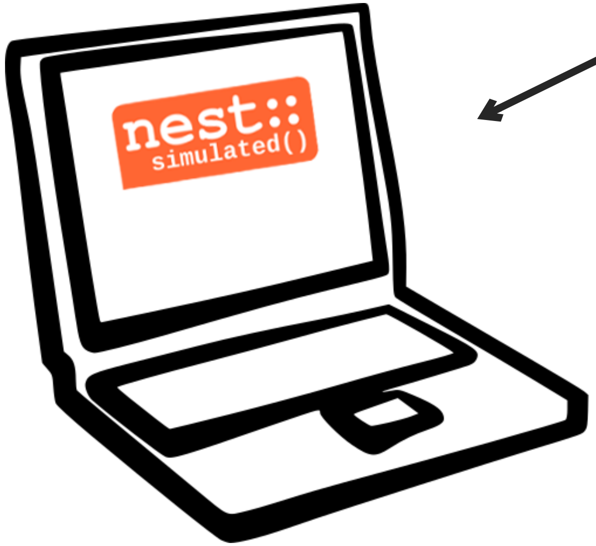
- NESTML is a **domain-specific modeling language** for the dynamical simulation of point neurons (spiking and rate-based), as well as synapses and synaptic plasticity rules.
  - Direct language support for differential equations, (spike) events, stochasticity, algorithms, ...
- Fully automated **simulation code generation** with solver benchmarking and selection

Why?

- **Reproducibility**: models are behaviorally validated in simulation runs
- Making computational neuroscience models **findable**, **accessible**, **interoperable**, and **reusable**
- Making **high-performance** spiking neural network simulation accessible

NESTML augments the simulation engine

nest::ml



NEST Desktop GUI

Python-standalone

NEST Simulator on HPC

SpiNNaker neuromorphic

NVIDIA

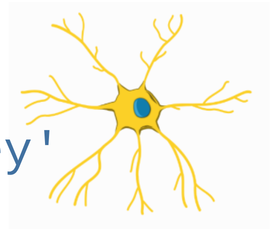
nest::gpu

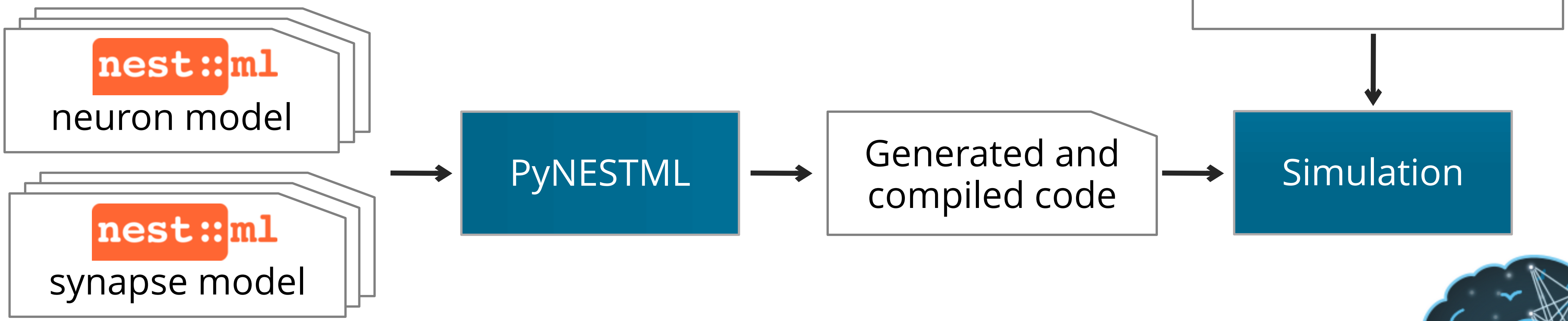
GPU-accelerated simulation

NESTML workflow for NEST

```
>>> import nest
>>> nest.Install('my_nestml_module')
>>> pre, post = nest.Create('my_hodgkin_huxley', 100)
>>> nest.Connect(pre, post,
     'all_to_all', syn_spec={'synapse_model': 'my_stdp'})
>>> nest.Simulate(1000)
```

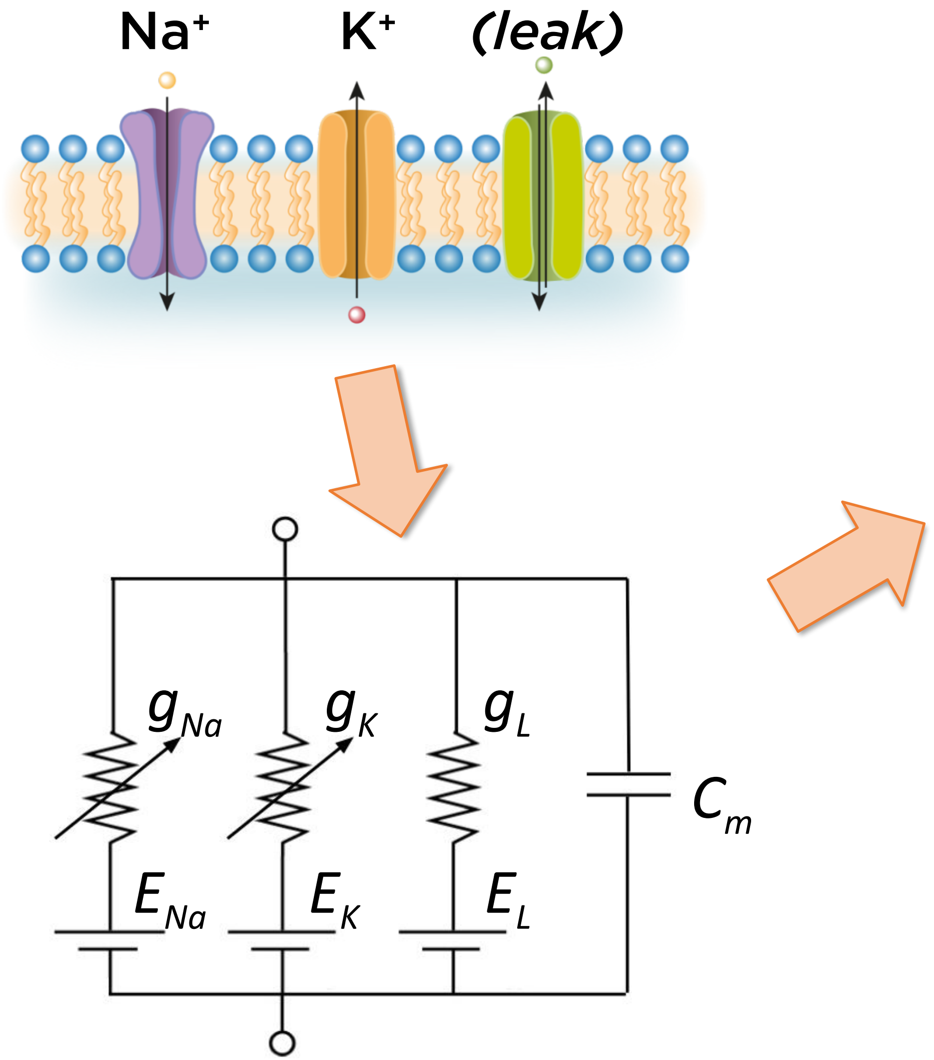
'my_hodgkin_huxley'





'my_stdp'

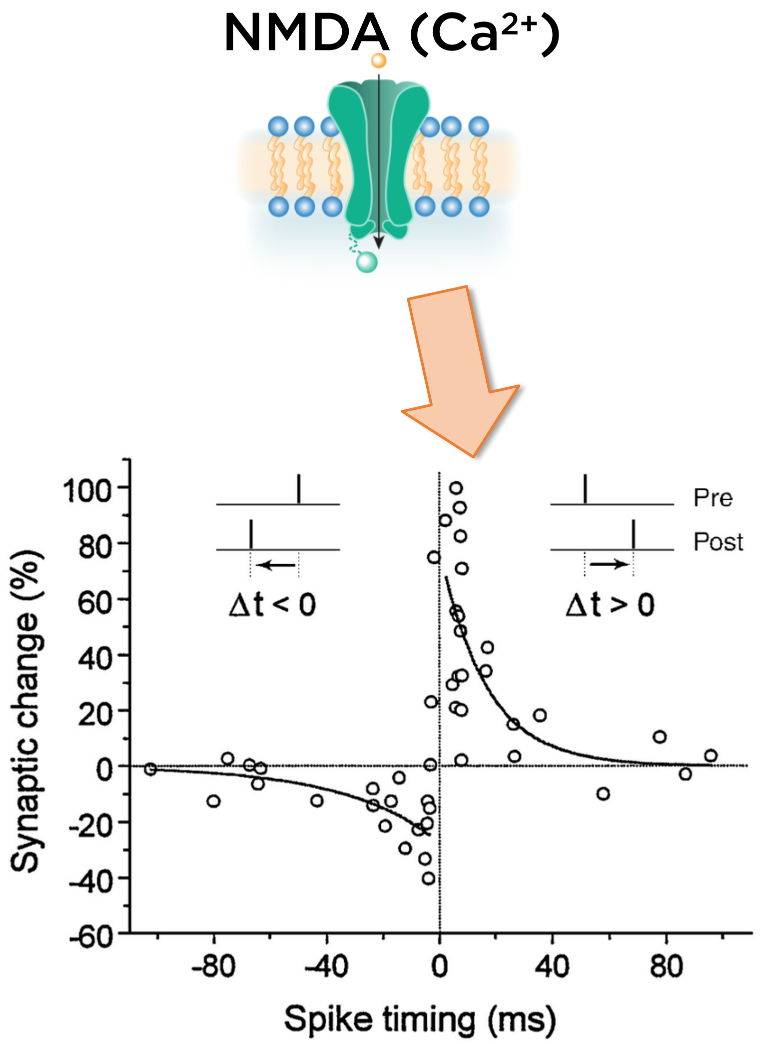
Neuron models in NESTML




```
neuron hodgkin_huxley:
  state:
    V_m mV = -65 mV     # membrane voltage
    Act_m, Act_n, Inact_h [...]

  equations:
    kernel syn_kernel = exp(-t / tau_syn)          # postsynaptic kernel
    inline I_Na pA = g_Na * Act_m**3 * Inact_h * (V_m - E_Na)
    inline I_K pA = g_K * Act_n**4 * (V_m - E_K)
    inline I_L pA = g_L * (V_m - E_L)
    Act_n' = (alpha_n(V_m) * (1 - Act_n) - beta_n(V_m) * Act_n) / ms
    Act_m' = (alpha_m(V_m) * (1 - Act_m) - beta_m(V_m) * Act_m) / ms
    Inact_h' = (alpha_h(V_m) * (1 - Inact_h) - beta_h(V_m) * Inact_h) / ms
    V_m' = -(I_Na + I_K + I_L) / C_m + convolve(syn_kernel, spikes)

  function alpha_n(V_m mV) real:
    return -0.05 * (V_m / mV + 34.) / (exp(-.1 * (V_m / mV + 34.)) - 1.)

  function alpha_m(V_m mV) real:
    [...]

  parameters:
    C_m pF = 250 pF
    V_threshold mV = 40 mV
    [...]

  update:
    integrate_odes()
    if V_m >= V_threshold:
      emit_spike()
```

Image credit: (c) S.B.C. Lehmann, FZ Jülich, (m) C. Linssen

Synaptic plasticity models in NESTML

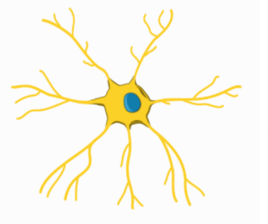


nest::ml

```
synapse stdp:
  state:
    w real = 1
    tr_post real = 0
    tr_pre real = 0

  equations:
    tr_pre' = -tr_pre / tau_tr
    tr_post' = -tr_post / tau_tr

  input:
    pre_spikes real <- spike
    post_spikes real <- spike

  onReceive(pre_spikes):
    w -= alpha * tr_post        # depress synapse
    tr_pre += 1                 # update presynaptic trace
    deliver_spike(w, delay)     # to postsynaptic partner

  onReceive(post_spikes):
    w += alpha * tr_pre         # potentiate synapse
    tr_post += 1                # update postsynaptic trace

  parameters:
    delay ms = 1 ms             # dendritic delay
    tau_tr ms = 50 ms           # pre/post trace time const.
    alpha real = .02            # learning rate
```

Image credit: (c) S.B.C. Lehmann, FZ Jülich, (m) C. Linssen; Bi & Poo 2001

```
synapse stdp:
  state:
    w real = 1
    tr_post real = 0
    tr_pre real = 0

  equations:
    tr_pre' = -tr_pre / tau_tr
    tr_post' = -tr_post / tau_tr

  input:
    pre_spikes real <- spike
    post_spikes real <- spike

  onReceive(pre_spikes):
    w -= alpha * tr_post        # depress synapse
    tr_pre += 1                 # update presynaptic trace
    deliver_spike(w, delay)     # to postsynaptic partner

  onReceive(post_spikes):
    w += alpha * tr_pre         # potentiate synapse
    tr_post += 1                # update postsynaptic trace

  parameters:
    delay ms = 1 ms             # dendritic delay
    tau_tr ms = 50 ms           # pre/post trace time const.
    alpha real = .02            # learning rate
```

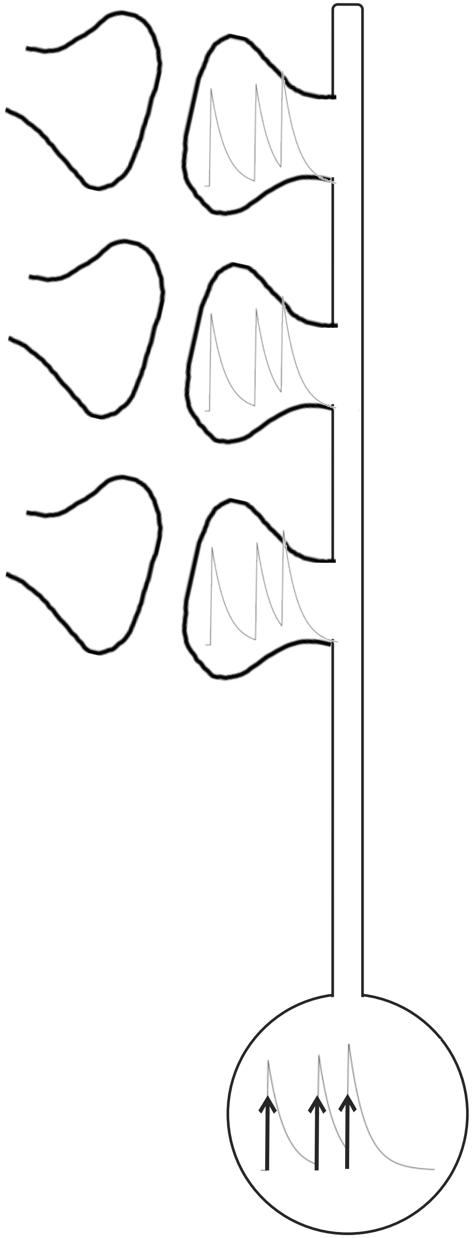


```
neuron hodgkin_huxley:
  state:
    V_m mV = -65 mV    # membrane voltage
    Act_m, Act_n, Inact_h [...]

  equations:
    kernel syn_kernel = exp(-t / tau_syn)
    inline I_Na pA = g_Na * Act_m**3 * [...]
    V_m' = -(I_Na + I_K + I_L) / C_m + [...]

  function alpha_n(V_m mV) real:
    return -0.05 * (V_m / mV + 34.) / (exp(-.1 *
(V_m / mV + 34.)) - 1.)

  parameters:
    C_m pF = 250 pF
    V_threshold mV = 40 mV
    [...]

  update:
    integrate_odes()
    if V_m >= V_threshold:
      emit_spike()
```

nest::ml

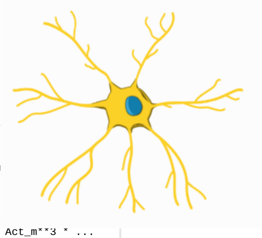
```
synapse stdp:
  state:
    w real = 1
    tr_post real = 0
    tr_pre real = 0

  equations:
    tr_pre' = -tr_pre / tau_tr
    tr_post' = -tr_post / tau_tr
  input:
    pre_spikes real <- spike
    post_spikes real <- spike

  onReceive(pre_spikes):
    w -= alpha * tr_post          # depress synapse
    tr_pre += 1                   # update presynaptic trace
    deliver_spike(w, delay)       # to postsynaptic partner
  onReceive(post_spikes):
    w += alpha * tr_pre           # potentiate synapse
    tr_post += 1                  # update postsynaptic trace
  parameters:
    delay ms = 1 ms               # dendritic delay
    tau_tr ms = 50 ms             # pre/post trace time const.
    alpha real = .02              # learning rate
```

Postsynaptic activity trace is specified in synapse model...



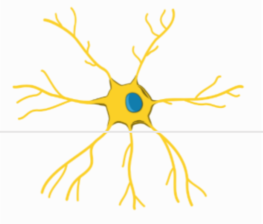
... but needs to be simulated as part of the neuron model to avoid redundant computations.

nest::ml



```
neuron hodgkin_huxley:
  state:
    V_m mV = -65 mV    # me
    Act_m, Act_n, Inact_h .

  equations:
    kernel syn_kernel = exp
    inline I_Na pA = g_Na * Act_m**3 * ...
    V_m' = -(I_Na + I_K + I_L) / C_m + ...

  function alpha_n(V_m mV) real:
    return -0.05 * (V_m / mV + 34.) /
(exp(-.1 * (V_m / mV + 34.)) - 1.)

  parameters:
    C_m pF = 250 pF
    V_threshold mV = 40 mV
    ...

  update:
    integrate_odes()
    if V_m >= V_threshold:
      emit_spike()
```

```
synapse stdp:
  state:
    w real = 1
    tr_post real = 0
    tr_pre real = 0

  equations:
    tr_pre' = -tr_pre / tau_tr
    tr_post' = -tr_post / tau_tr

  input:
    pre_spikes real <- spike
    post_spikes real <- spike

  onReceive(pre_spikes):
    w -= alpha * tr_post        # depress synapse
    tr_pre += 1                 # update presynaptic trace
    deliver_spike(w, delay)     # to postsynaptic partner

  onReceive(post_spikes):
    w += alpha * tr_pre         # potentiate synapse
    tr_post += 1                # update postsynaptic trace

  parameters:
    delay ms = 1 ms             # dendritic delay
    tau_tr ms = 50 ms           # pre/post trace time const.
    alpha real = .02            # learning rate
```



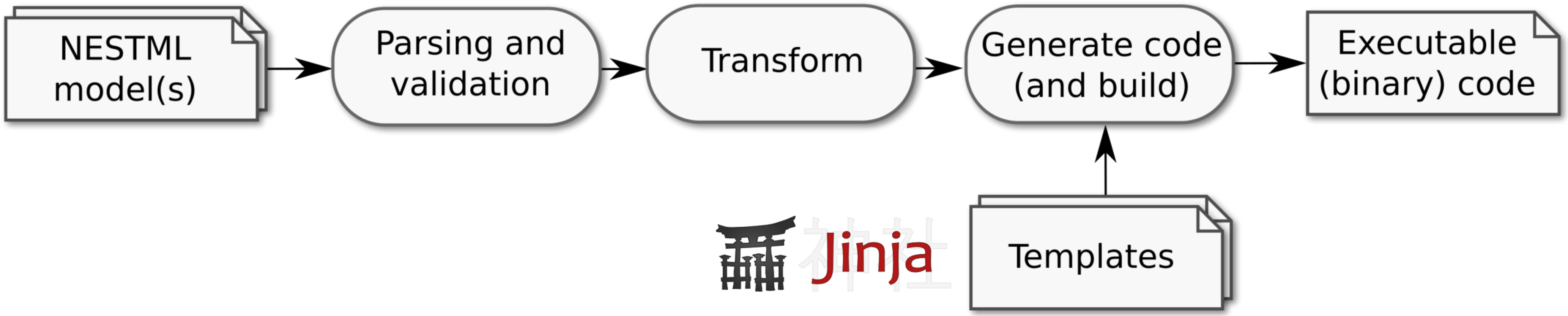
```
neuron hodgkin_huxley:
  state:
    V_m mV = -65 mV     # membrane voltage
    Act_m, Act_n, Inact_h [...]

  equations:
    kernel syn_kernel = exp(-t / tau_syn)
    inline I_Na pA = g_Na * Act_m**3 * [...]
    V_m' = -(I_Na + I_K + I_L) / C_m + [...]

  function alpha_n(V_m mV) real:
    return -0.05 * (V_m / mV + 34.) / (exp(-.1 *
(V_m / mV + 34.)) - 1.)

  parameters:
    C_m pF = 250 pF
    V_threshold mV = 40 mV
    [...]

  update:
    integrate_odes()
    if V_m >= V_threshold:
      emit_spike()
```

PyNESTML toolchain: modular and extensible

- Jinja templates are used for code generation. The toolchain is modular and written in Python, which, taken together, allows for a great deal of flexbility and ease of customizing and adding novel code generation platform targets.

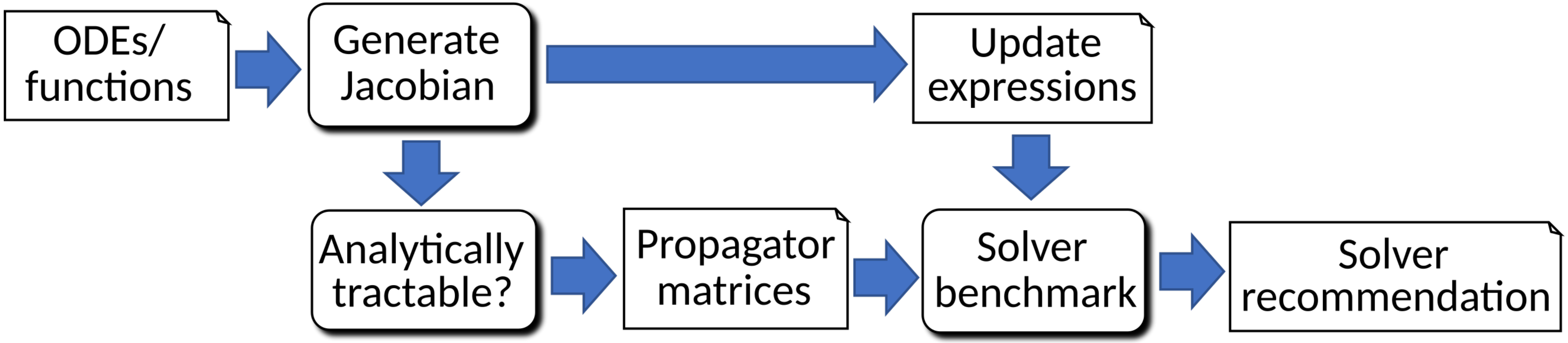
Example snippet from synapse template for Python target:

```
{%- if parameter_syms_with_iv|length > 0 %}
    # initial values for parameters
{%- filter indent(4) %}
{%- for parameter in parameter_syms_with_iv %}
{%-   with variable = parameter %}
{%-     include "directives/MemberInitialization.jinja2" %}
{%-   endwith %}
{%- endfor %}
{%- endfilter %}
{%- endif %}
```



Automatic selection and generation of integration schemes for systems of ODEs

- Fully automated symbolic analysis of systems of ODEs using sympy
- Generates ”propagator” solver for dynamics that admits an analytic solution
- Numeric solver benchmarking and recommendation



Advanced synaptic plasticity rules

Neuromodulation

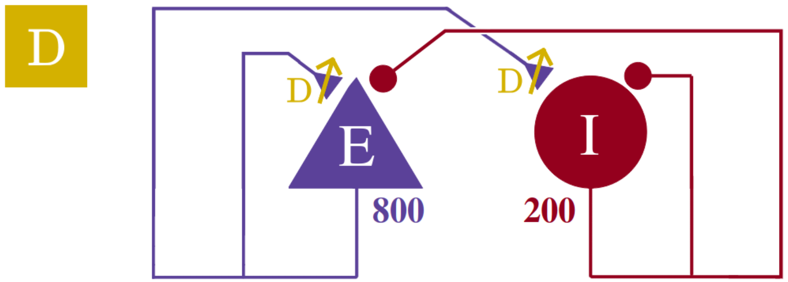
Spatially diffuse neuromodulators like dopamine can flexibly be used as “third factors” in synaptic plasticity rules to impement, reinforcement learning.

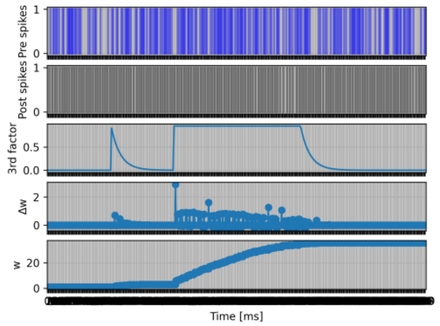
Postsynaptic dendritic current-modulated STDP

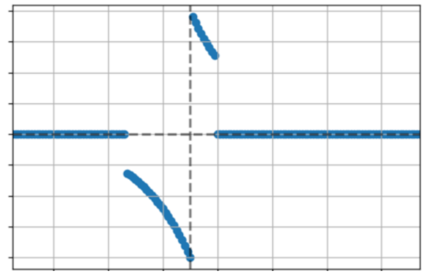
Arbitrary postsynaptic quantities, such as (active) dendritic currents, can be used to modulate synaptic plasticity.

Triplet and non-linear STDP

Higher-order STDP rules, such as the triplet rule, as well as non-linear STDP variants, involving for example the clamping of a weight change outside a given window, can be easily and flexibly implemented using the NESTML syntax.





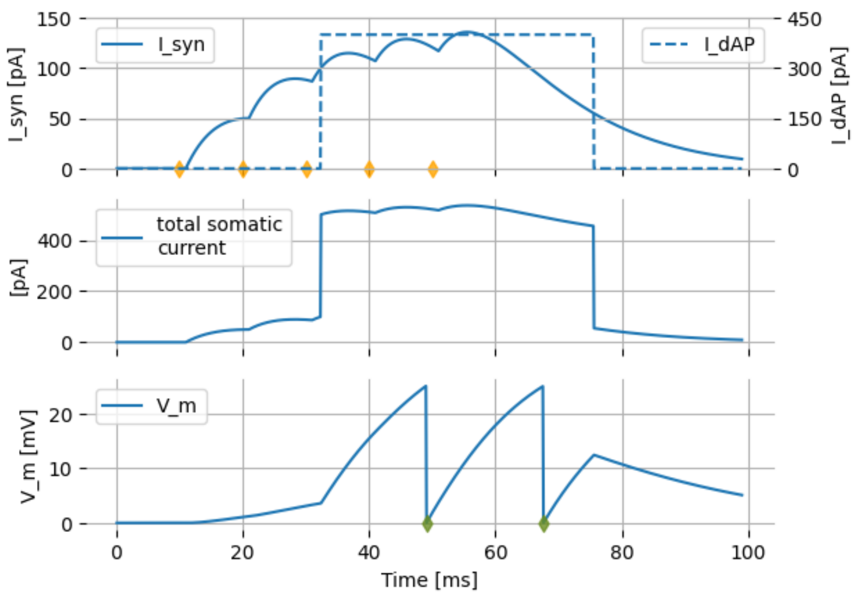


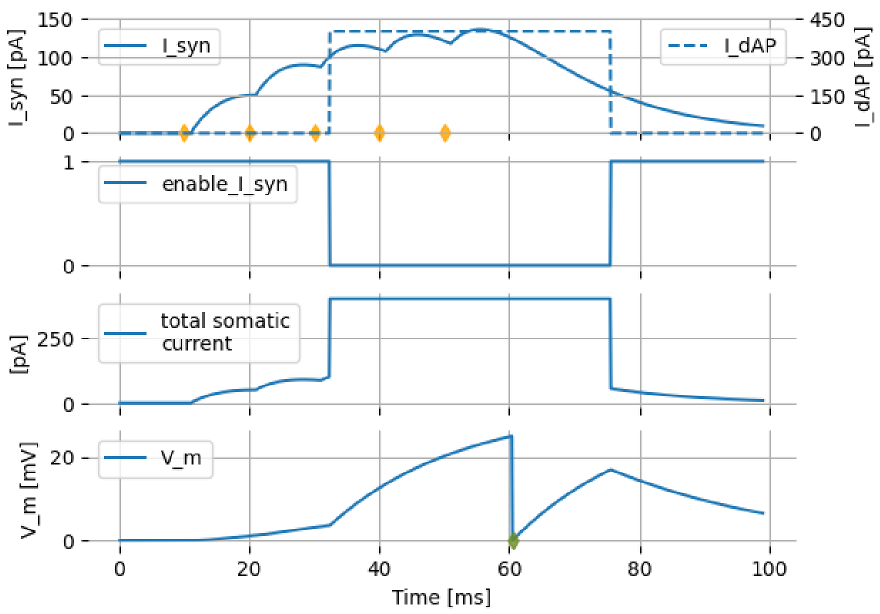
NESTML interactive tutorials

Active dendrite

"Nonlinear" or "active" dendritic compartment, that can, independently from the soma, generate a dendritic action potential.

Shows subtle differences in postsynaptic dendritic integration.

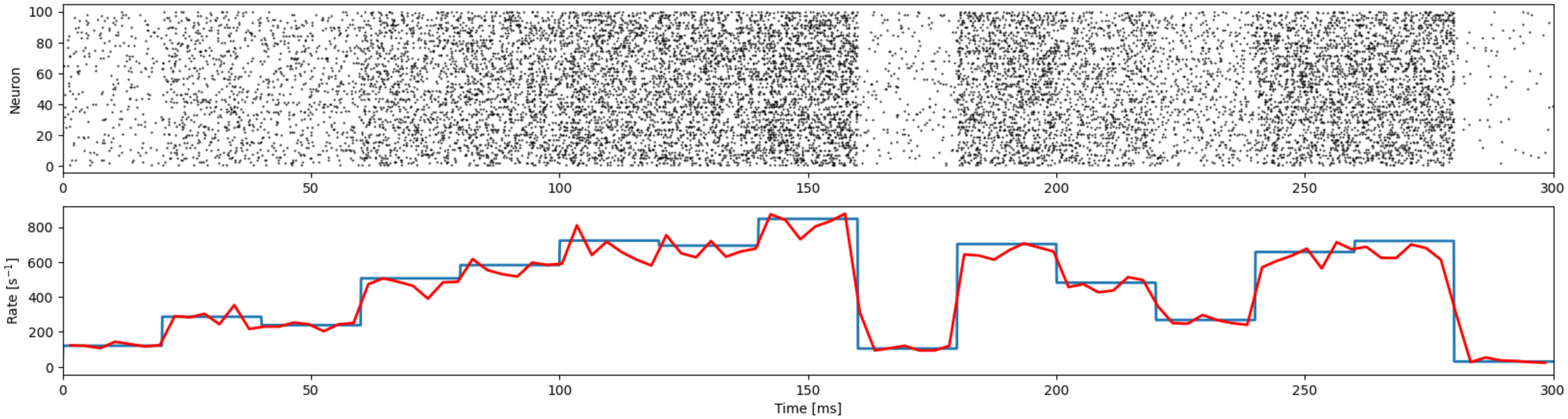




NESTML interactive tutorials

Inhomogeneous Poisson generator

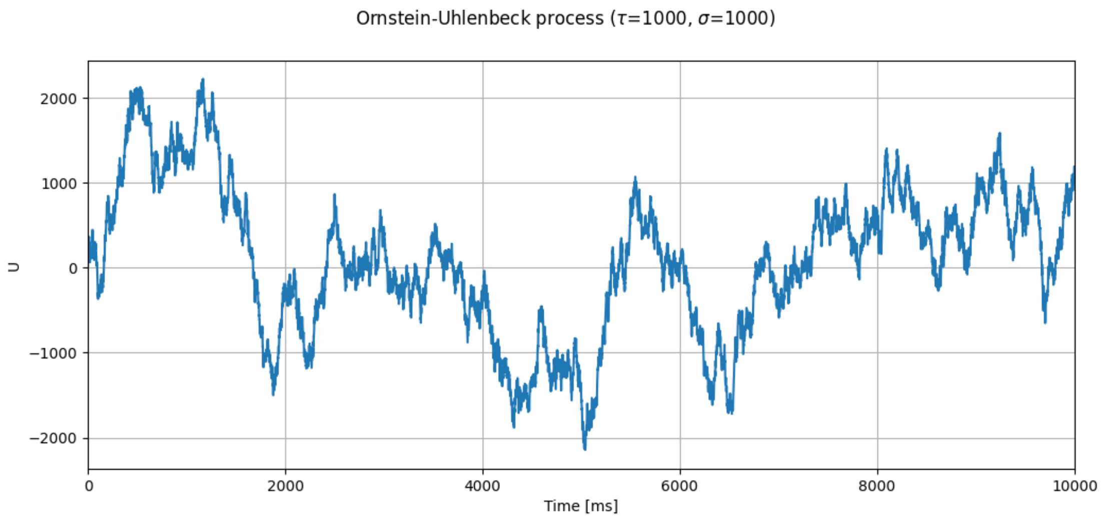
NESTML model for an inhomogeneous Poisson process. The rate of the model is piecewise constant and is defined by an array containing desired rates (in units of 1/s) and an array of equal length containing the corresponding times (in units of ms).

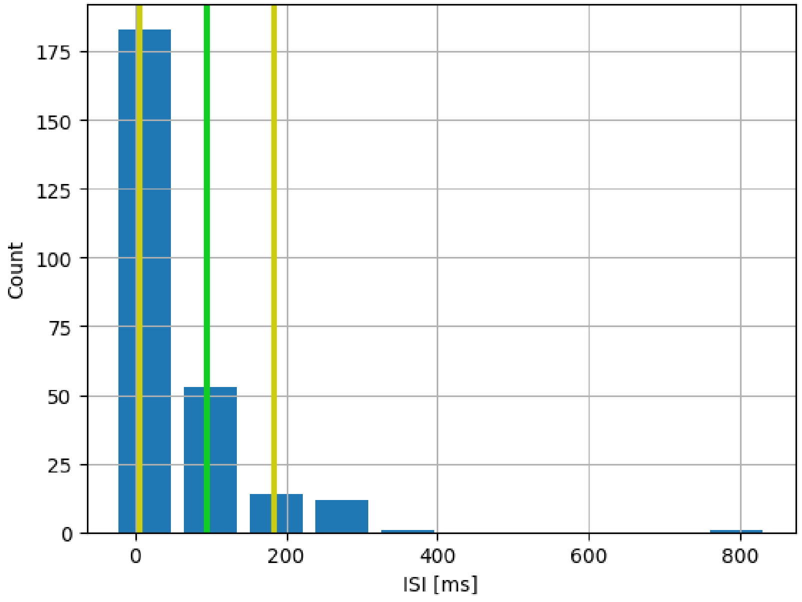


NESTML interactive tutorials

Ornstein-Uhlenbeck noise

The Ornstein-Uhlenbeck process is often used as a source of noise because it is well understood and has convenient properties (it is a Gaussian process, has the Markov property, and is stationary).

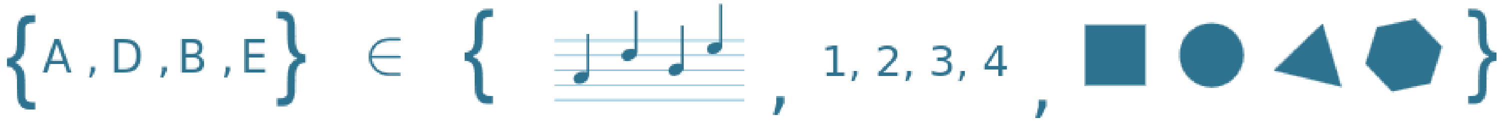


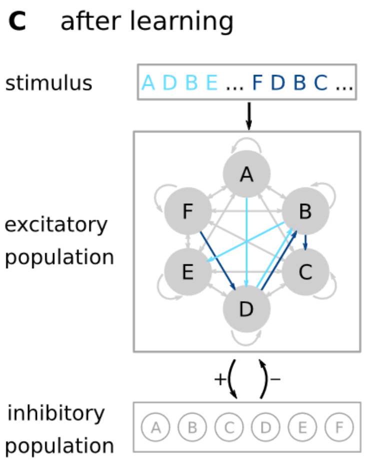


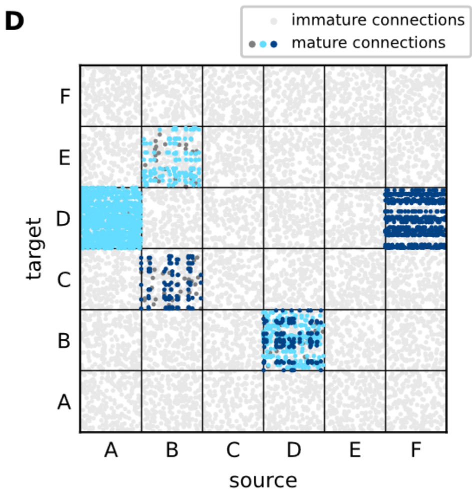
NESTML interactive tutorials

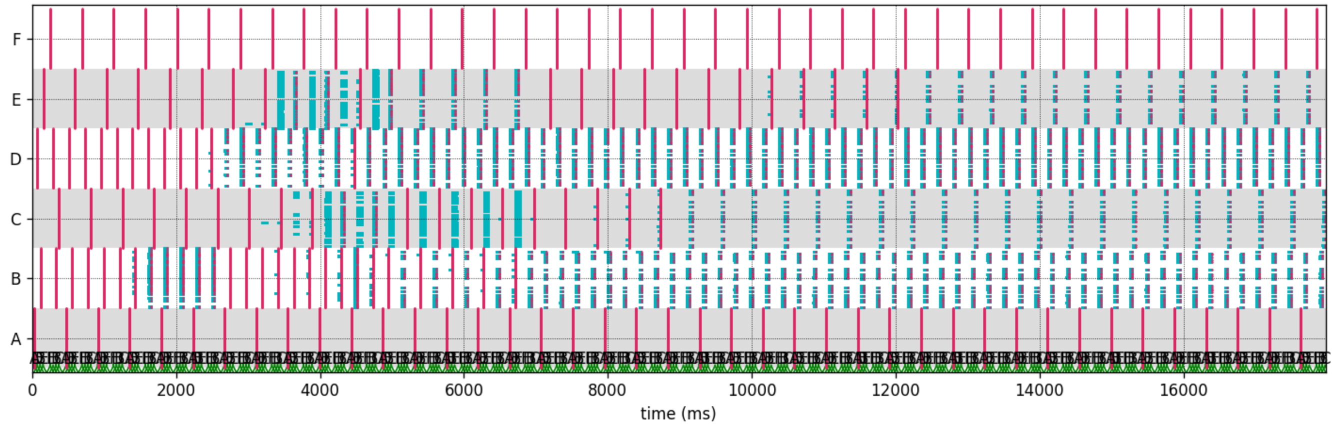
Sequence learning and replay

In this tutorial, a neuron and synapse model are defined in NESTML that are subsequently used in a network to perform learning, prediction and replay of sequences of items, such as letters, images or sounds





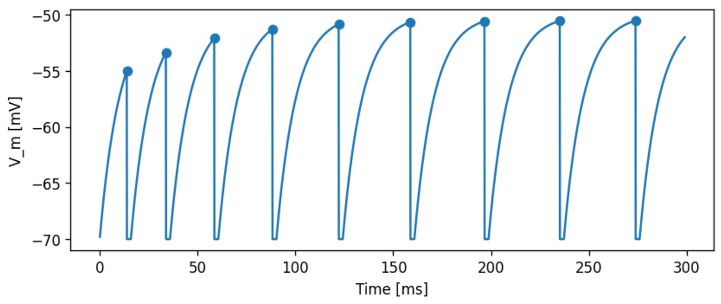


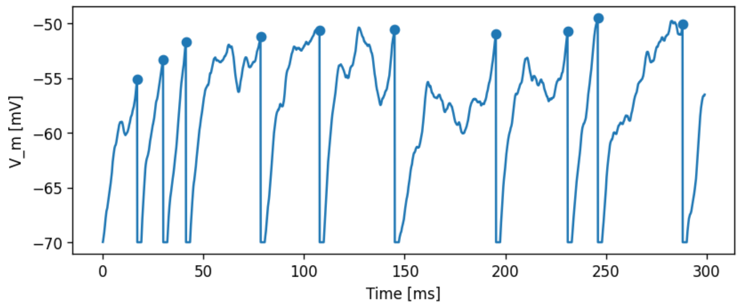


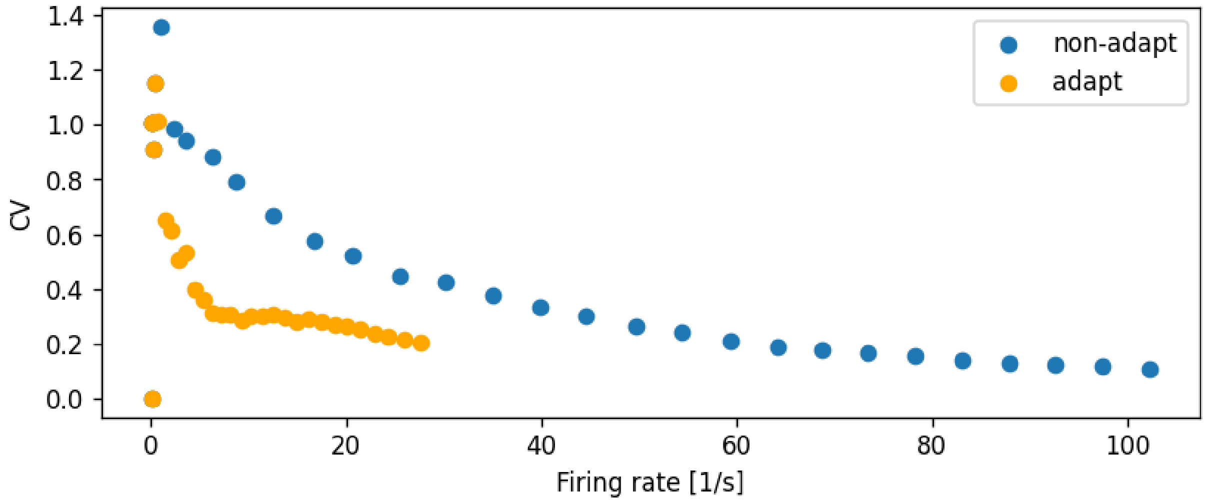
NESTML interactive tutorials

Spike-frequency adaptation

In this tutorial, we will go step by step through adding two different types of SFA mechanism to a neuron model, threshold adaptation and an adaptation current, and then evaluate how the new models behave in simulation.



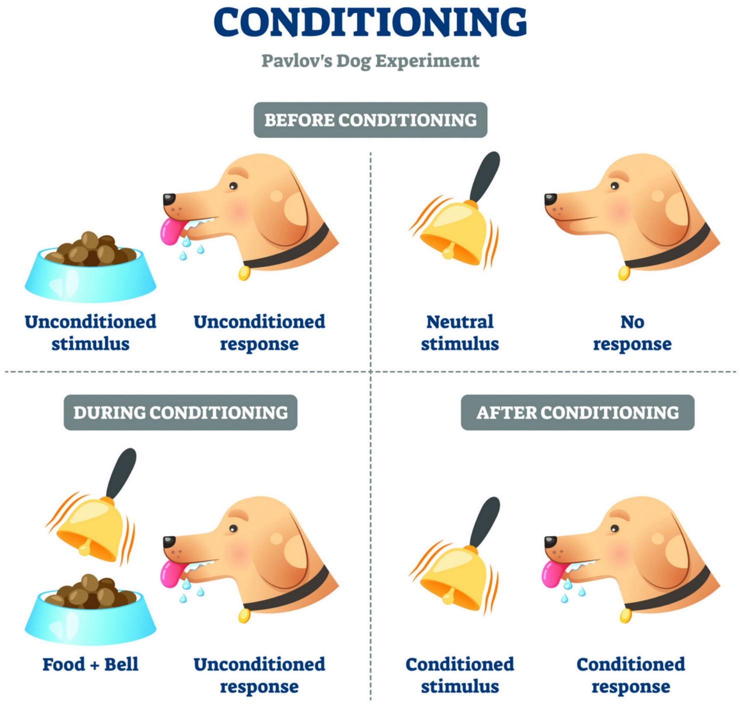


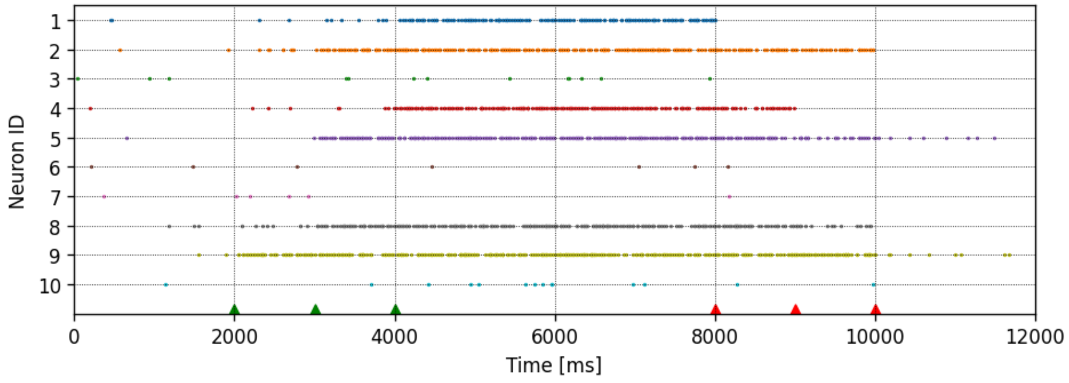


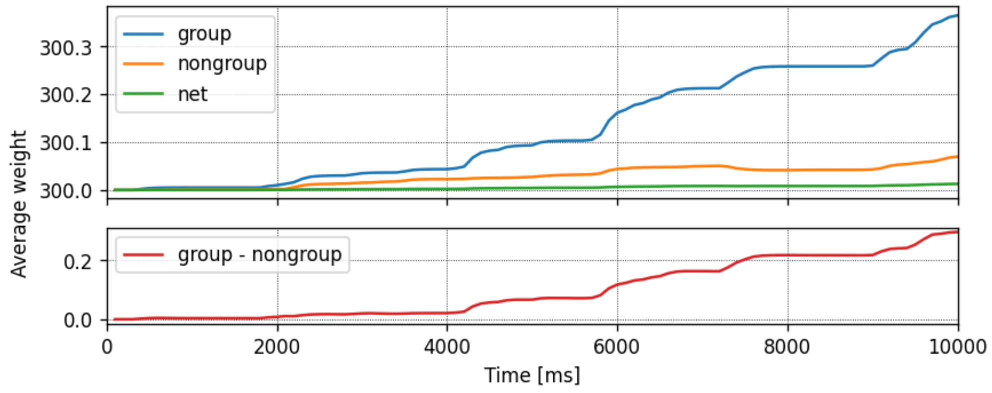
NESTML interactive tutorials

Dopamine-modulated STDP synapse

Pavlov and Thompson (1902) first described classical conditioning: a phenomenon in which a biologically potent stimulus–the Unconditional Stimulus (UC)—is initially paired with a neutral stimulus—the Conditional Stimulus (CS). After many trials, learning is observed when the previously neutral stimuli start to elicit a response similar to that which was previously only elicited by the biologically potent stimulus.



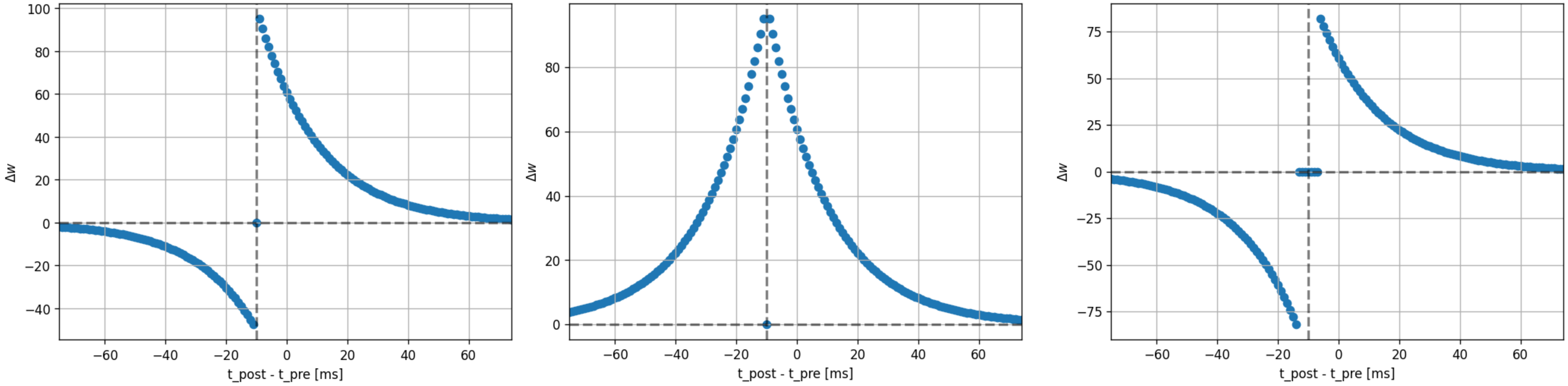




NESTML interactive tutorials

STDP windows

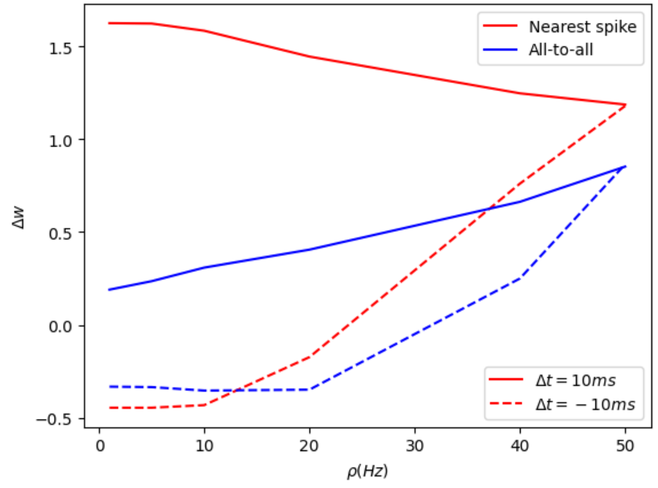
In this tutorial, we will plot the "window function", relating the weight change of a synapse to the relative timing of a single pair of pre- and postsynaptic spikes. We will then use NESTML to explore different variants of the STDP rule.

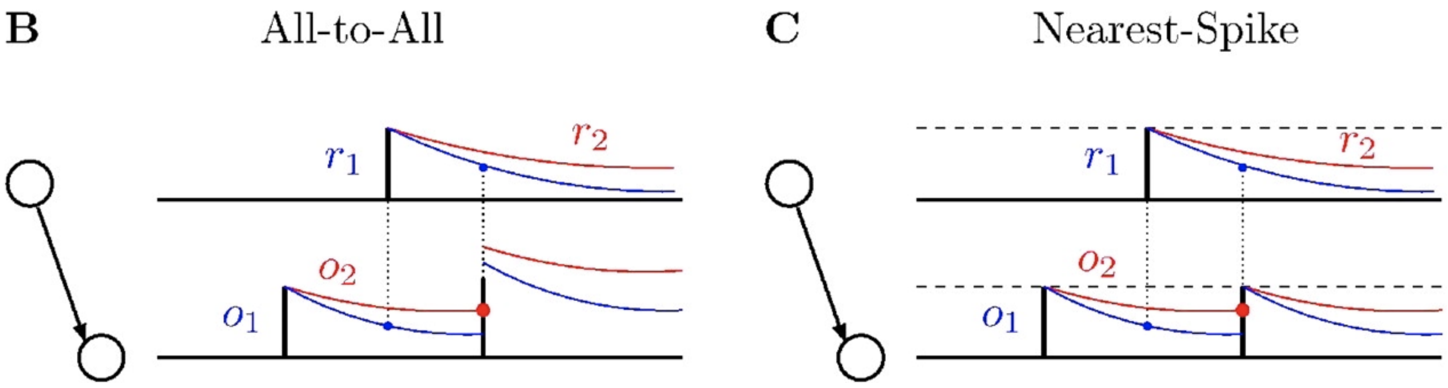


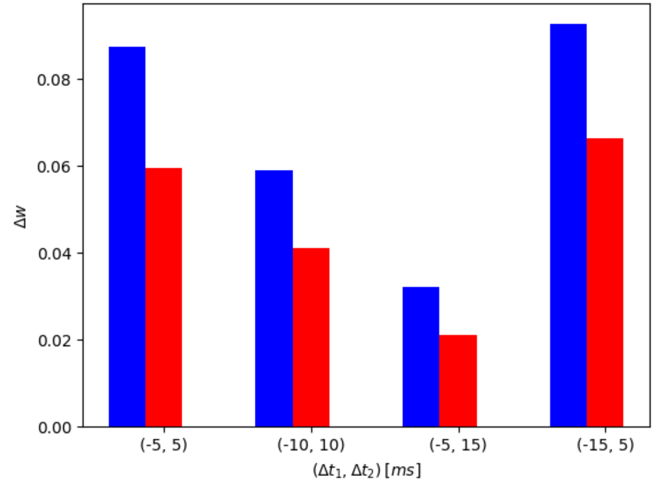
NESTML interactive tutorials

Triplet STDP synapse

The triplet STDP rule considers sets of three spikes, i.e., two presynaptic and one postsynaptic spikes or two postsynaptic and one presynaptic spike, allowing it to capture statistically higher-order correlations.







NESTML uses best practices in software engineering

- Automated testing (CI): models are behaviorally validated in simulation runs

- Extensive documentation and automated HTML documentation generation for models in the extensive neuron and synapse models library:

  https://nestml.readthedocs.org/

- Open development:

  https://github.com/nest/nestml

  GNU GPL v2.0 licensed

Thank you!

Jochen M. Eppler
Abigail Morrison
Markus Diesmann
Konstantin Perun
Pooja Babu

Dimitri Plotnikov
Inga Blundell
Tanguy Fardet
Jessica Mitchell
Sara Konradi
Ayssar Benelhedi

... and to all our users!

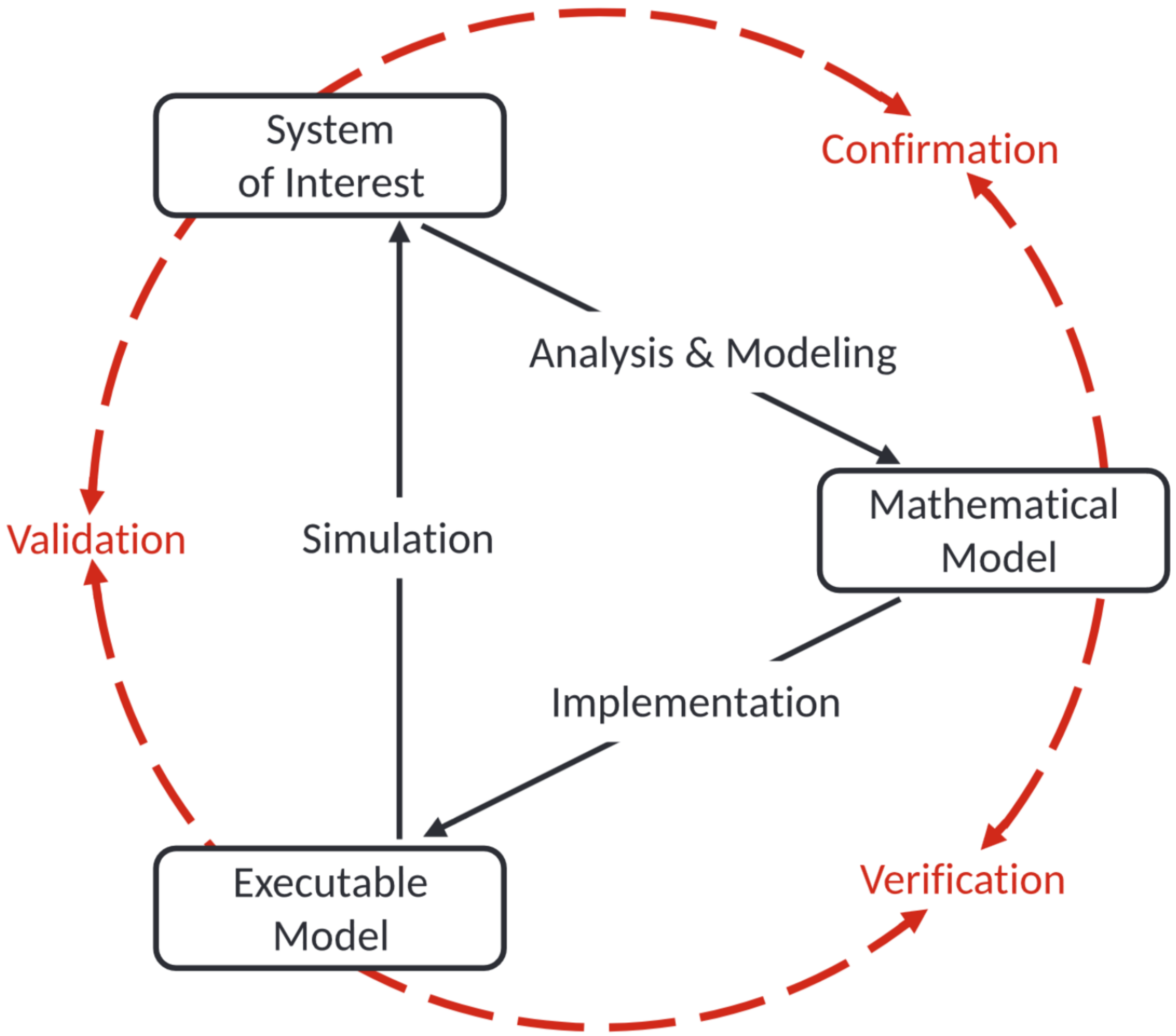
EBRAINS

JÜLICH
FORSCHUNGSZENTRUM

This software was initially supported by the JARA-HPC Seed Fund *NESTML - A modeling language for spiking neuron and synapse models for NEST* and the Initiative and Networking Fund of the Helmholtz Association and the Helmholtz Portfolio Theme *Simulation and Modeling for the Human Brain*.

This software was developed in part or in whole in the Human Brain Project, funded from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreements No. 720270, No. 785907 and No. 945539 (Human Brain Project SGA1, SGA2 and SGA3).

Workflow


System of Interest
Confirmation
Analysis & Modeling
Validation
Simulation
Mathematical Model
Implementation
Executable Model
Verification

Further reading

NEST Simulator:

https://nest-simulator.readthedocs.io/

NESTML:

https://nestml.readthedocs.io/

NEST Desktop:

https://nest-desktop.readthedocs.io/