NESTML Tutorial

Charl Linssen & Jochen M. Eppler

Installing NESTML

Running NESTML

NESTML is built on top of MontiCore, which is built on top of Java8 and requires Maven to be built.
NESTML requires a very recent version of SymPy to analyze the equations

NESTML is (currently) a bit complicated to use

1. *Install everything from scratch. See instructions on the NESTML GitHub page*
2. *Use the Docker container on a Linux machine*
3. *Use our virtual machine with everything pre-installed*

Running the NESTML application

/home/username/nestml/models

Files

- models
  - iaf_psc_alpha.nestml

shell

```
cd ~/nestml/docker
h run ../models
```

Files

- models
  - iaf_psc_alpha.nestml
  - build
    - iaf_psc_alpha_neuron{.cpp,.h}
    - models{.cpp,.h}
    - CMakeList.txt

NESTML application

*A helper script to **provision** and **run** the container*

*`nestml_docker.sh` takes the **command** as an argument and creates/runs the container with the current release*

If `--from_sources` is given, the most recent sources from GitHub shall be used.

shell

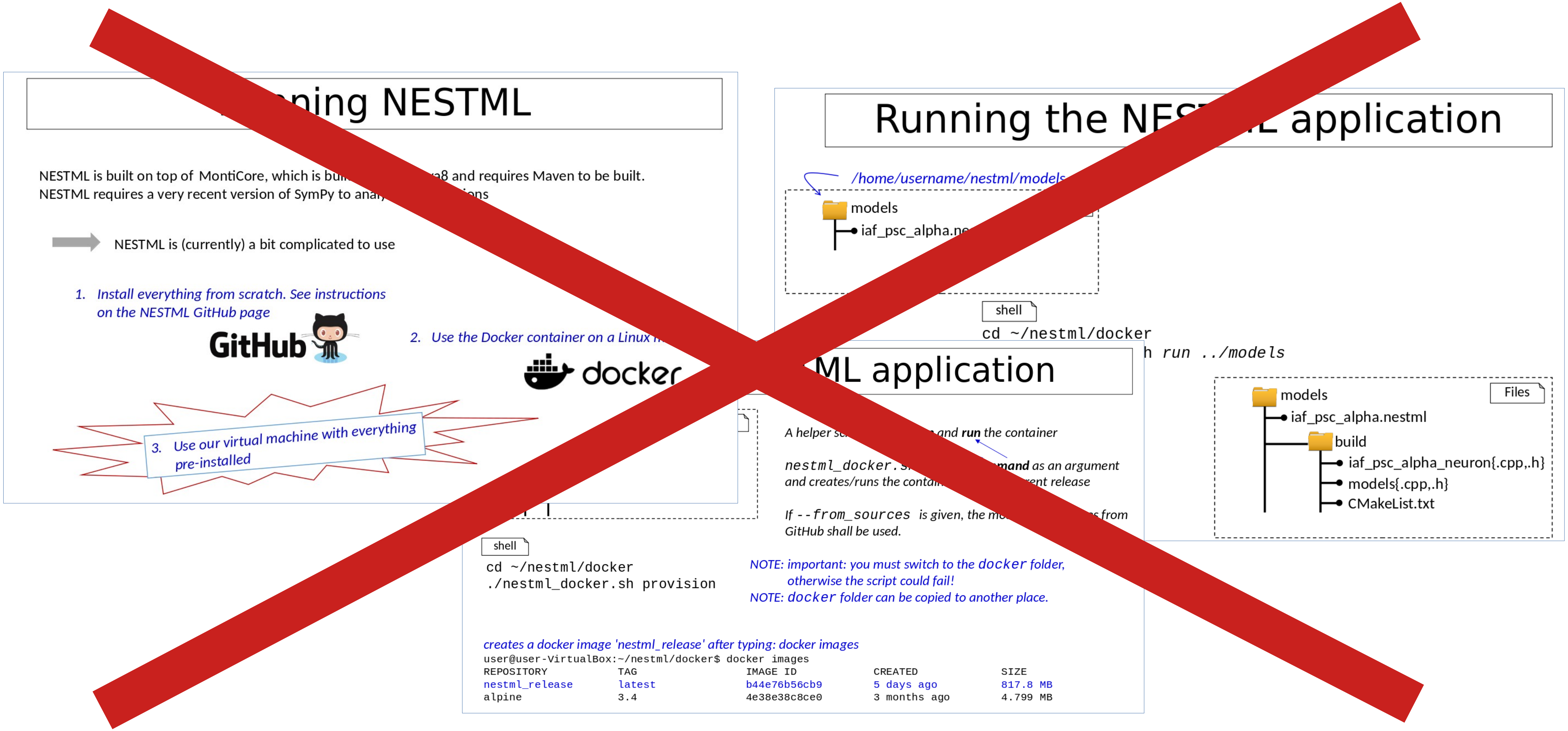
```
cd ~/nestml/docker
./nestml_docker.sh provision
```

NOTE: important: you must switch to the `docker` folder, otherwise the script could fail!
NOTE: `docker` folder can be copied to another place.

creates a docker image 'nestml_release' after typing: docker images

```
user@user-VirtualBox:~/nestml/docker$ docker images
REPOSITORY          TAG            IMAGE ID         CREATED          SIZE
nestml_release      latest         b44e76b56cb9     5 days ago       817.8 MB
alpine              3.4            4e38e38c8ce0     3 months ago     4.799 MB
```

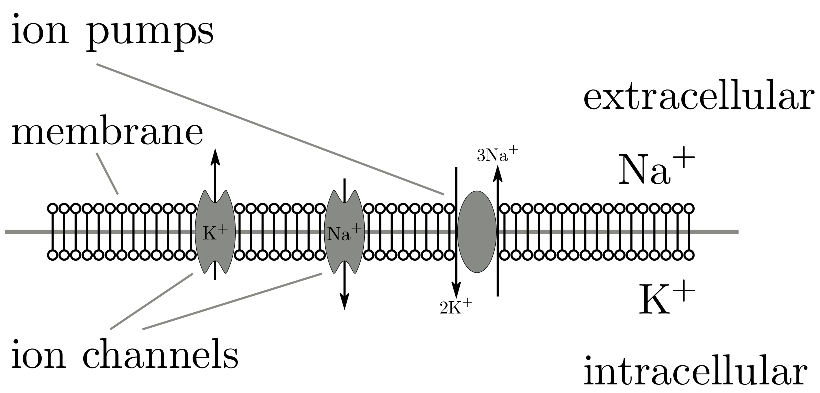
Installing NESTML

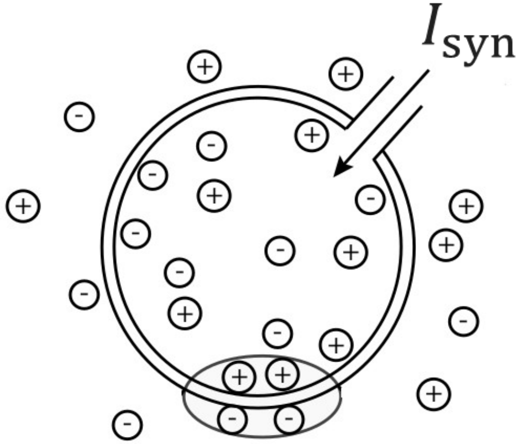


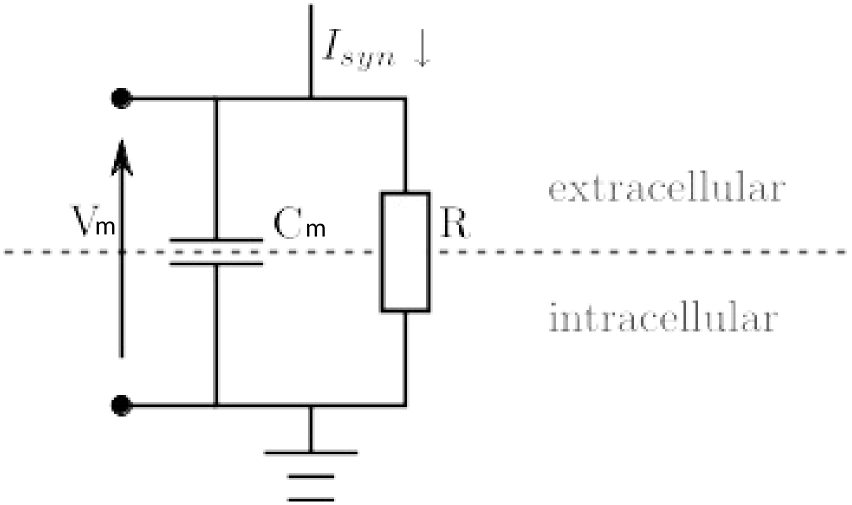
Installing NESTML

```
pip3 install nestml
```

Modelling biological neurons







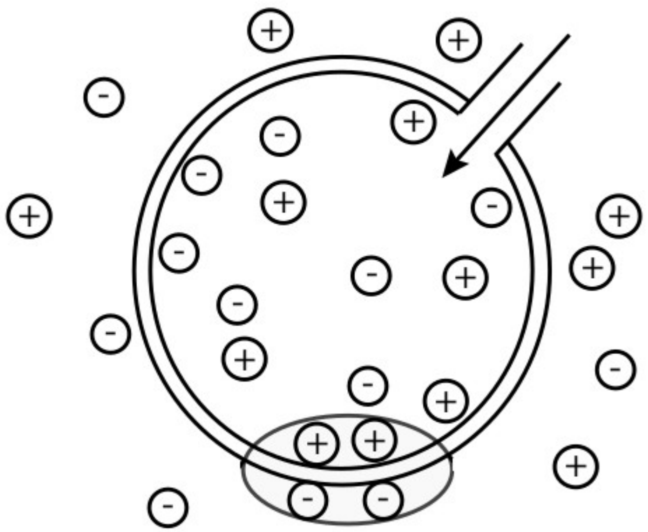
$$\frac{d\,V_{\text{m}}}{dt} = -\frac{V_{\text{m}}}{\tau_{\text{m}}} + \frac{I_{\text{syn}}}{C_{\text{m}}}$$

$$\tau_{\text{m}} = \text{C}_{\text{m}} \cdot \text{R}$$

Mapping biological neurons to NESTML

Model

neuron-model based on an RC-circuit



NESTML

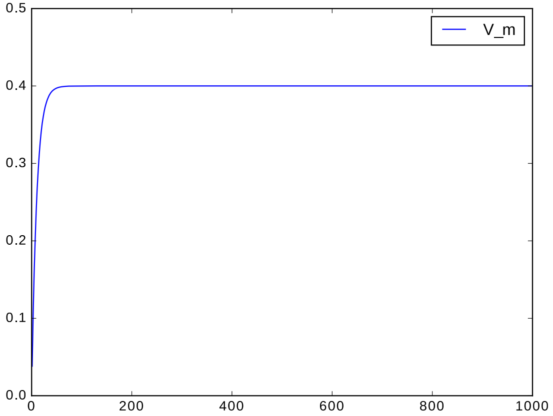
<<rc_neuron.nestml>>

```
neuron rc_neuron:

end
```

Mapping biological neurons to NESTML

Model

$$\frac{d\,V_m}{dt} = -\frac{V_m}{\tau_m} + \frac{I_{syn}}{C_m}$$



NESTML

<<rc_neuron.nestml>>

```
neuron rc_neuron:

  initial_values:
    V_m mV = 0mV
  end

  equations:
    V_m' = -V_m/tau_m + I_syn/C_m
  end

  parameters:
    # values taken from experiments
    C_m pF   = 250pF
    tau_m ms = 10ms
    I_syn pA = 10pA
  end

  update:
    integrate_odes()
  end

end
```

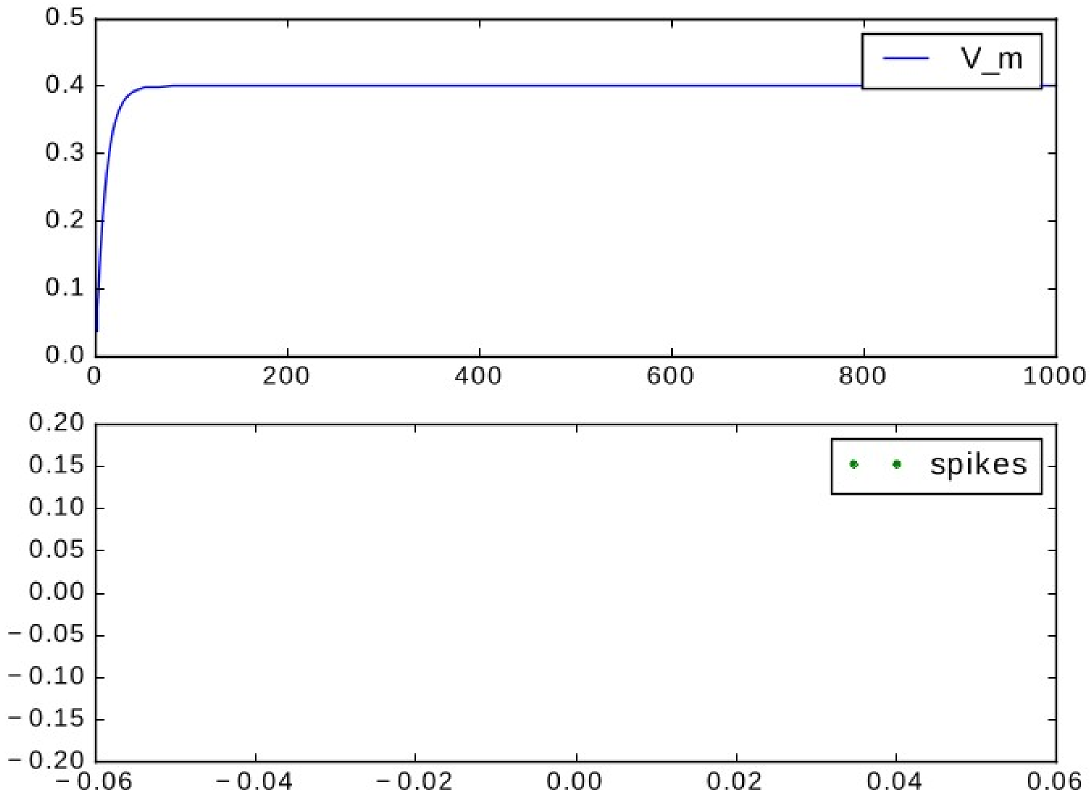
Simulating `rc_neuron`

NEST
<<Runtime>>

- Simulating `rc_neuron` for 1000ms with constant input current of 10pA

→ Strictly positive membrane potential

→ No spikes



Adding the resting potential E_L

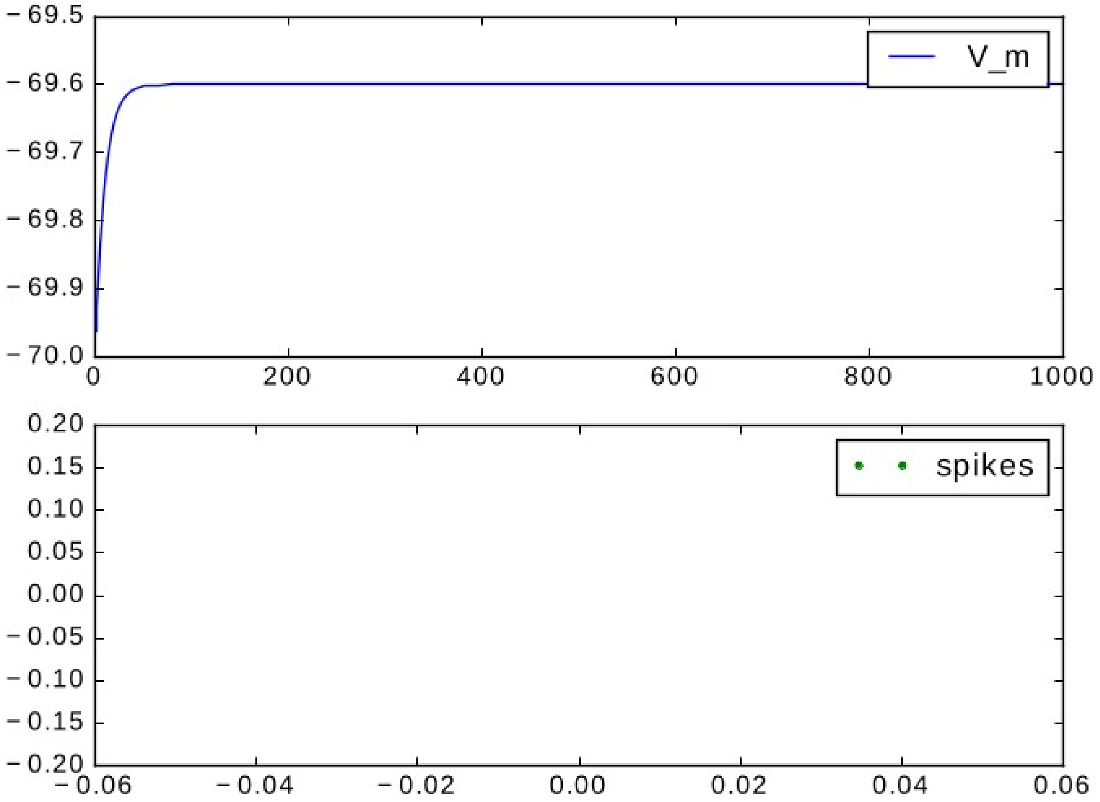
NEST
<<Runtime>>

- Shift V_m by E_L:

NESTML
<<rc_neuron_rest.nestml>>

```
neuron rc_neuron_rest:
  initial_values:
    V_m mV = E_L
  end

  equations:
    V_m' = -(V_m-E_L)/tau_m + I_syn/C_m
  end

  parameters:
    E_L mV = -70mV
  end

  ...
end
```

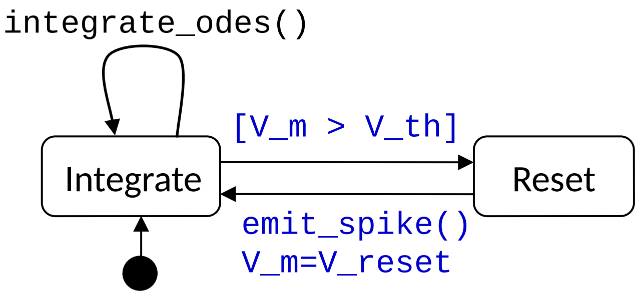
→ Still no spikes



Spiking and reset

Model

AD



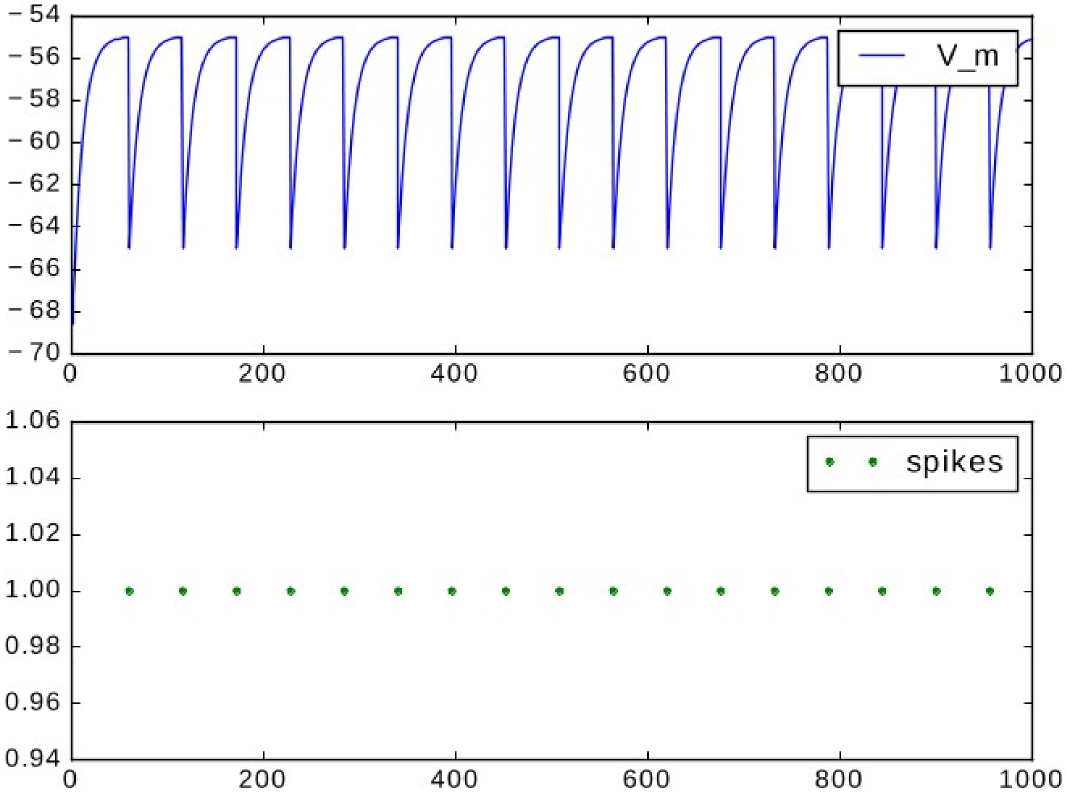
NESTML

<<rc_fire.nestml>>

```
neuron rc_fire:
  parameters:
    V_th mV = -55mV - E_L
  end
  update:
    integrate_odes()
    if V_m > V_th:
      V_m = V_reset
      emit_spike()
    end
  end
  ...
end
```

NEST

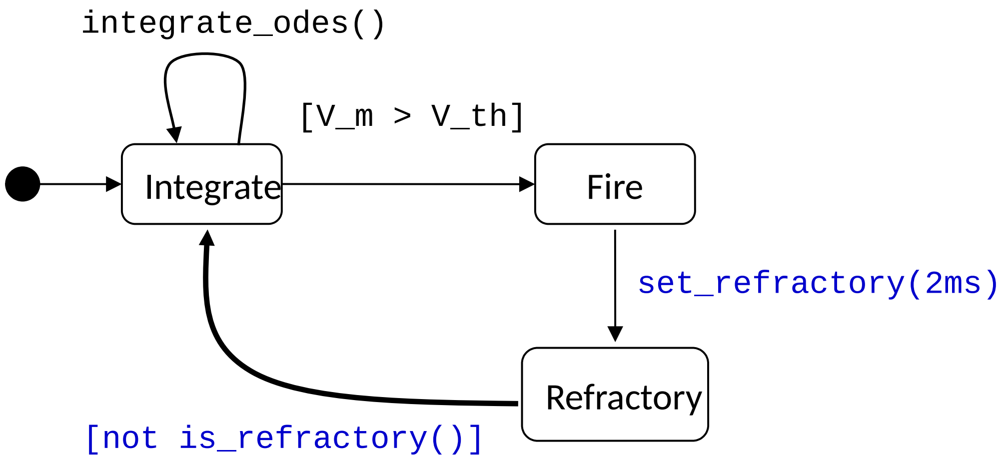
<<Runtime>>



Refractoriness

Model

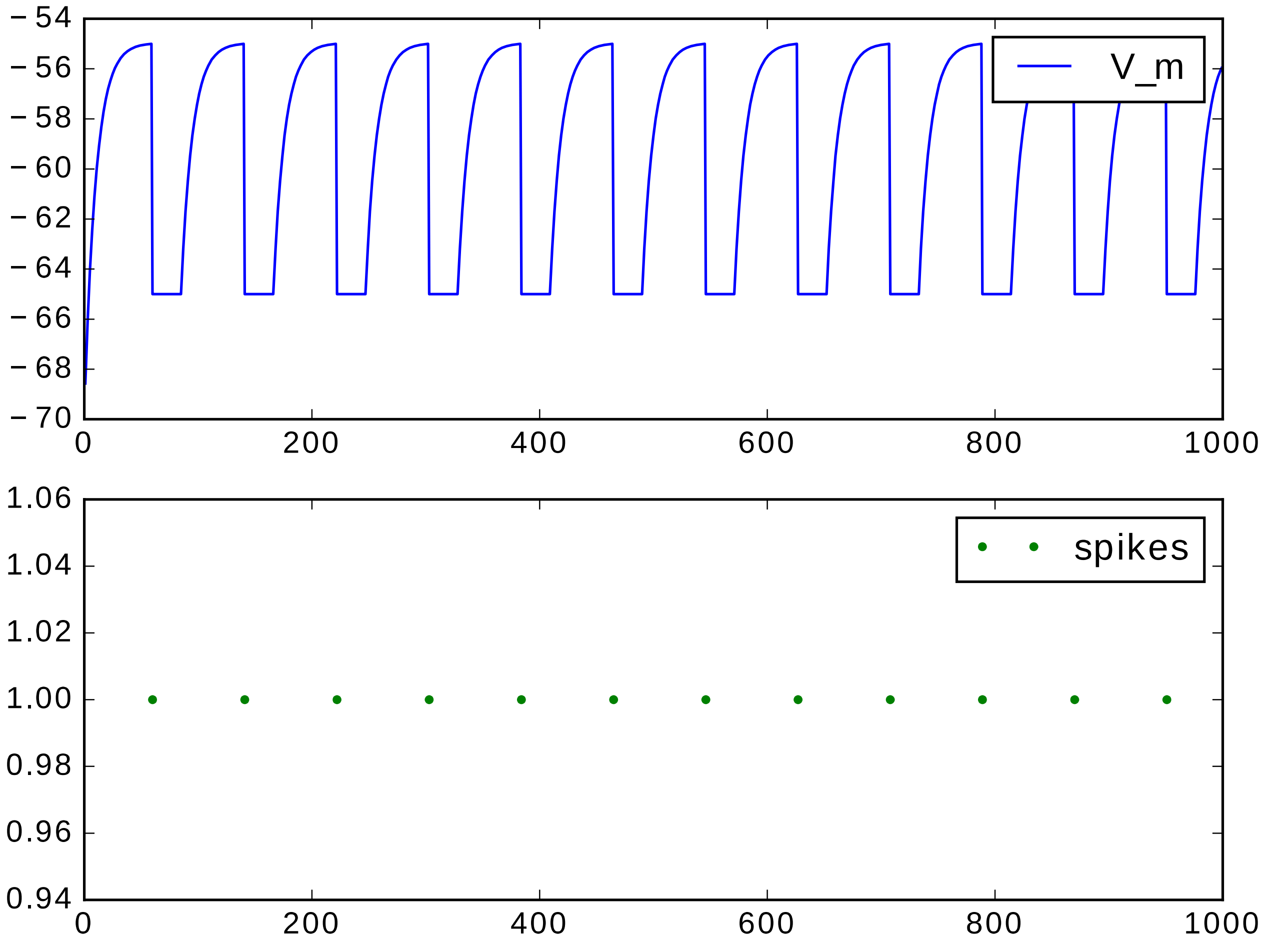
AD



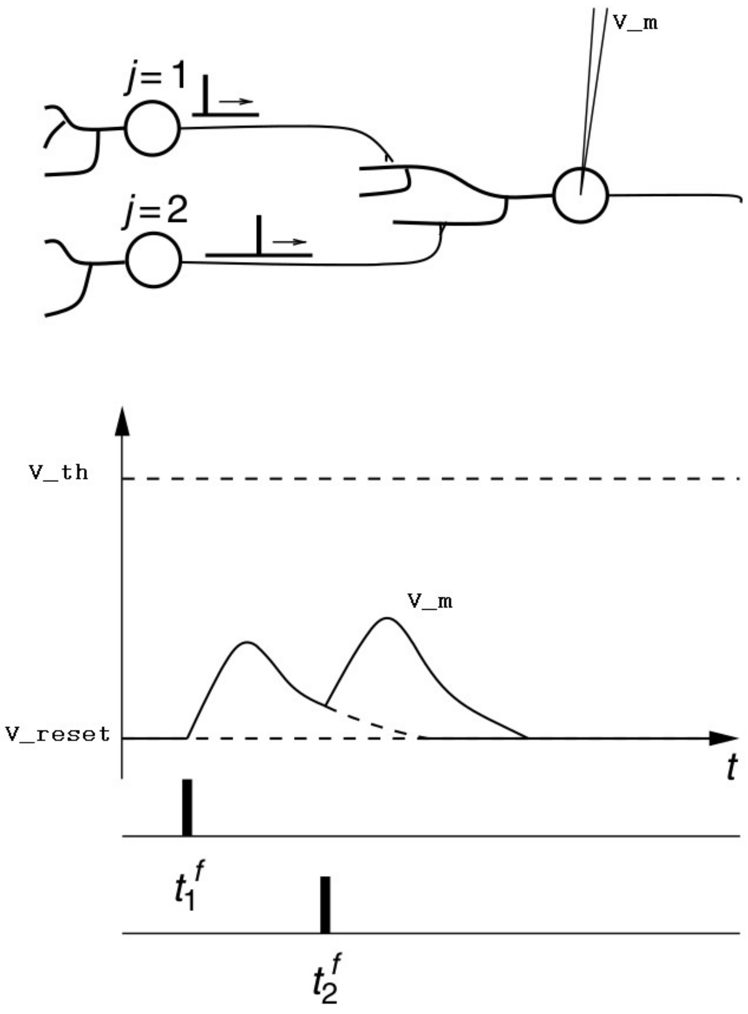
NESTML
<<rc_refractory>>
...

```
neuron rc_refractory:
  parameters:
    ref_counts integer = 0
    ref_timeout ms = 2ms
  end
  internals:
    timeout_ticks integer = steps(ref_timeout)
  end
  update:
    if ref_counts == 0:
      integrate_odes()
      if V_m > V_th:
        emit_spike()
        ref_counts = timeout_ticks
        V_m = V_reset
      end
    else:
      ref_counts -= 1
    end
  end
end
```

Simulating `rc_refractory`



Input handling

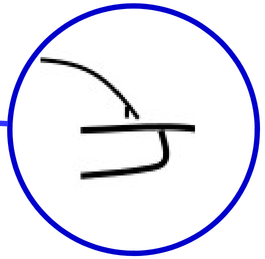


(Source: Wulfram Gerstner, Werner M. Kistler, Richard Naud, Liam Paninski-Neuronal Dynamics From Single Neurons to Networks and Models of Cognition)

Spike input

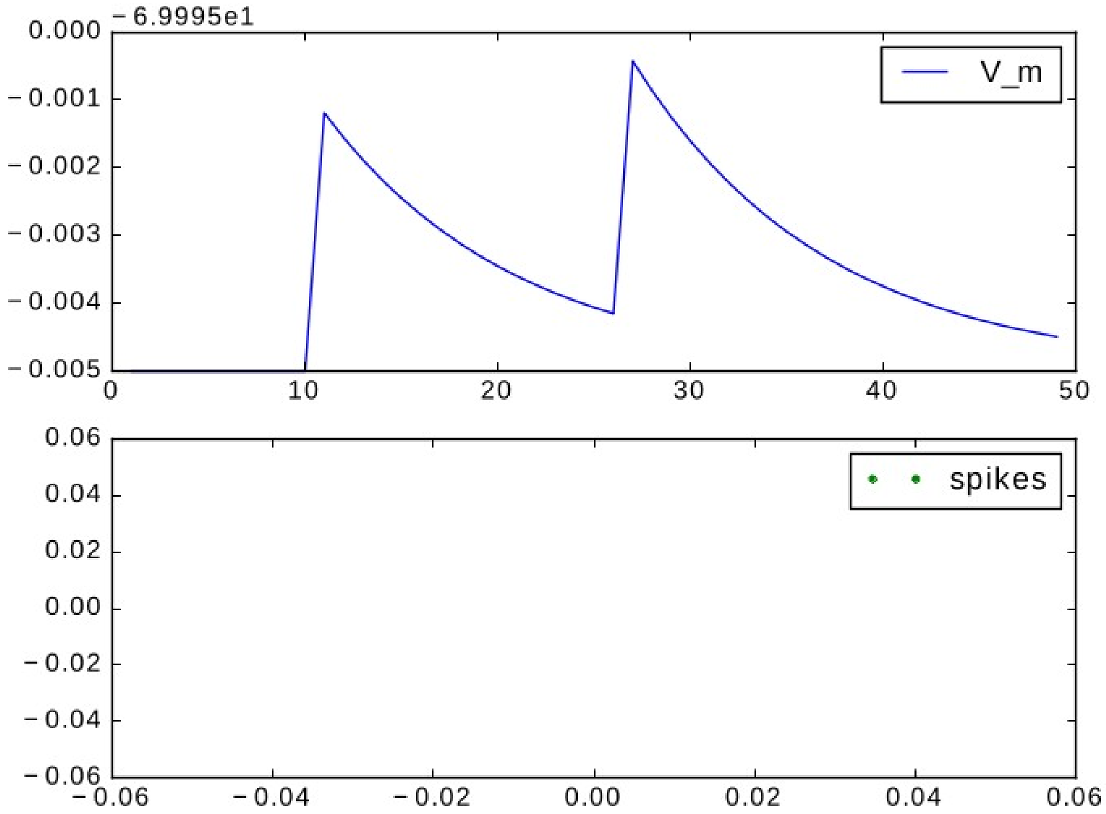
```
neuron rc_input:
  initial_values:
    V_m mV = E_L
  end

  equations:
    V_m' = -(V_m-E_L)/tau_m + I_syn/C_m
  end

  parameters:
    E_L mV = -70mV
    ...
  end

  input:
    I_syn pA <- spike
  end

  output: spike

end
```

NESTML
<<rc_input>>



buffer can be `inhibitory` , `excitatory` or both (if nothing else stated)

NEST
<<Runtime>>



Synaptic response

```
neuron rc_alpha_response:
  initial_values:
    V_m mV = E_L
    I_a real = 0
    I_a' 1/ms = e/tau_syn
  end


  equations:
    shape I_a'' = (-2/tau_syn) * I_a'-(1/tau_syn**2) * I_a
    V_m' = -(V_m-E_L)/tau_m + convolve(I_a, spikes)/C_m
  end

  input:
    spikes pA <- spike
  end

  output: spike

  update:
    integrate_odes()
    ...
  end

end
```

NESTML

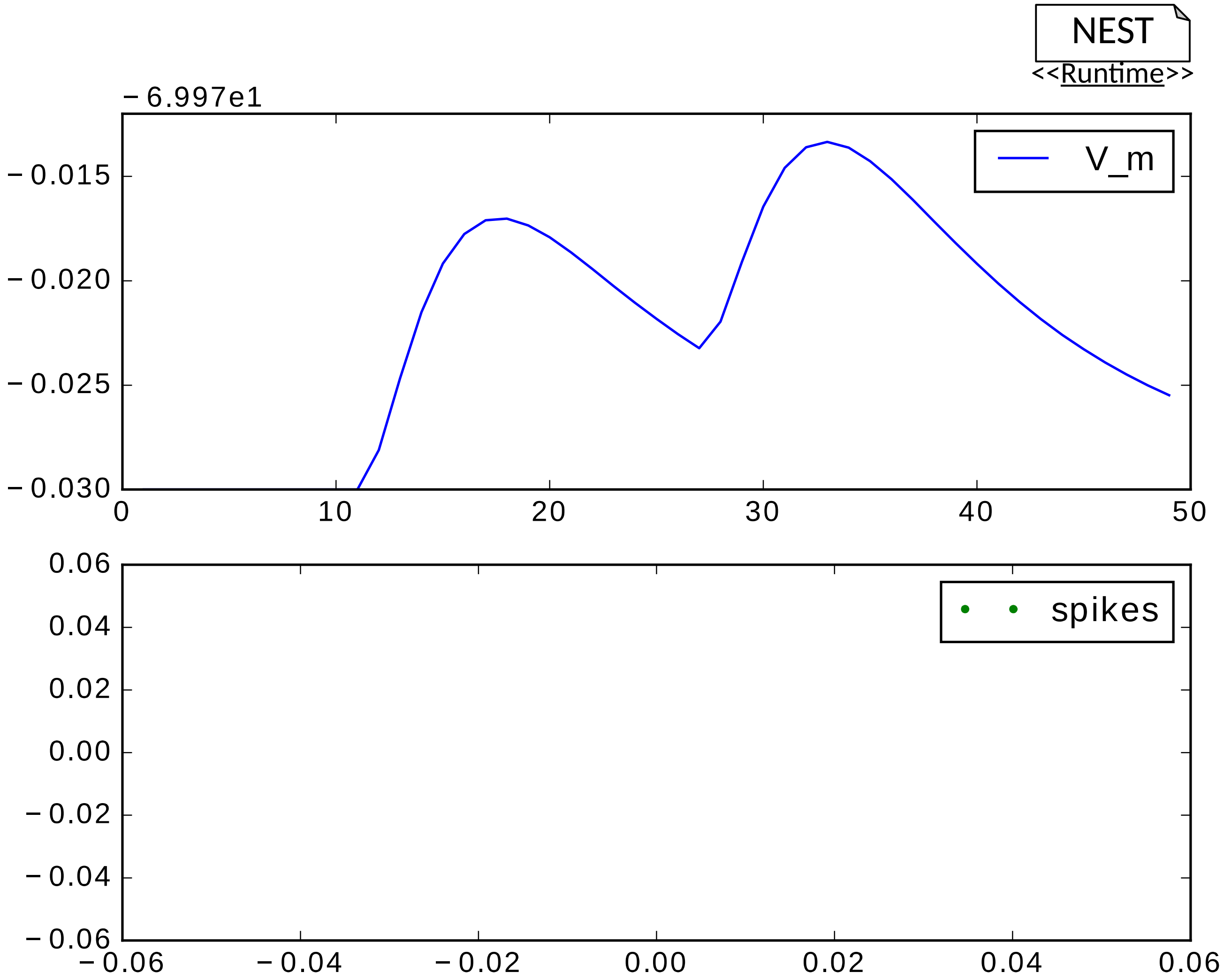
<<rc_alpha>>

ODEs of order n require all initial values of the derivatives from 0 to n-1

$$\sum_{t_i \leq t, i \in \mathbb{N}} \sum_{w \in W} w \cdot I_\mathrm{a}(t_i - t)$$

$$= \sum_{t_i \leq t, i \in \mathbb{N}} I_\mathrm{a}(t_i - t) \sum_{w \in W} w$$

Simulating `rc_alpha_response`



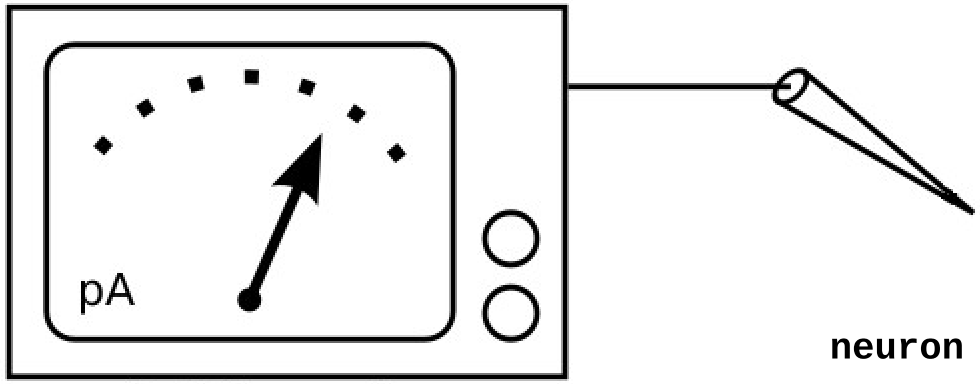
Shape notation

NESTML
<<rc_shape>>

initial values
computed automatically

```
neuron rc_alpha_response_shape:
  state:
    V_m mV = E_L

  end



  equations:
    shape I_a = (e/tau_syn) * t * exp(-t/tau_syn)
    V_m' = -(V_m-E_L)/tau_m + convolve(I_a, spikes)/C_m
  end

  input:
    spikes pA <- spike
  end

  output: spike

  update:
    integrate_odes()
    ...
  end


end
```

Injecting currents



DC Generator

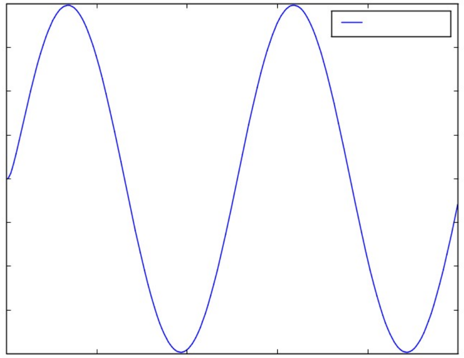
PyNEST

```
currents = nest.Create('ac_generator', 1,
                       {'amplitude': 100.0,
                        'frequency': 2.0})
nest.Connect(currents , rc_currents)
```

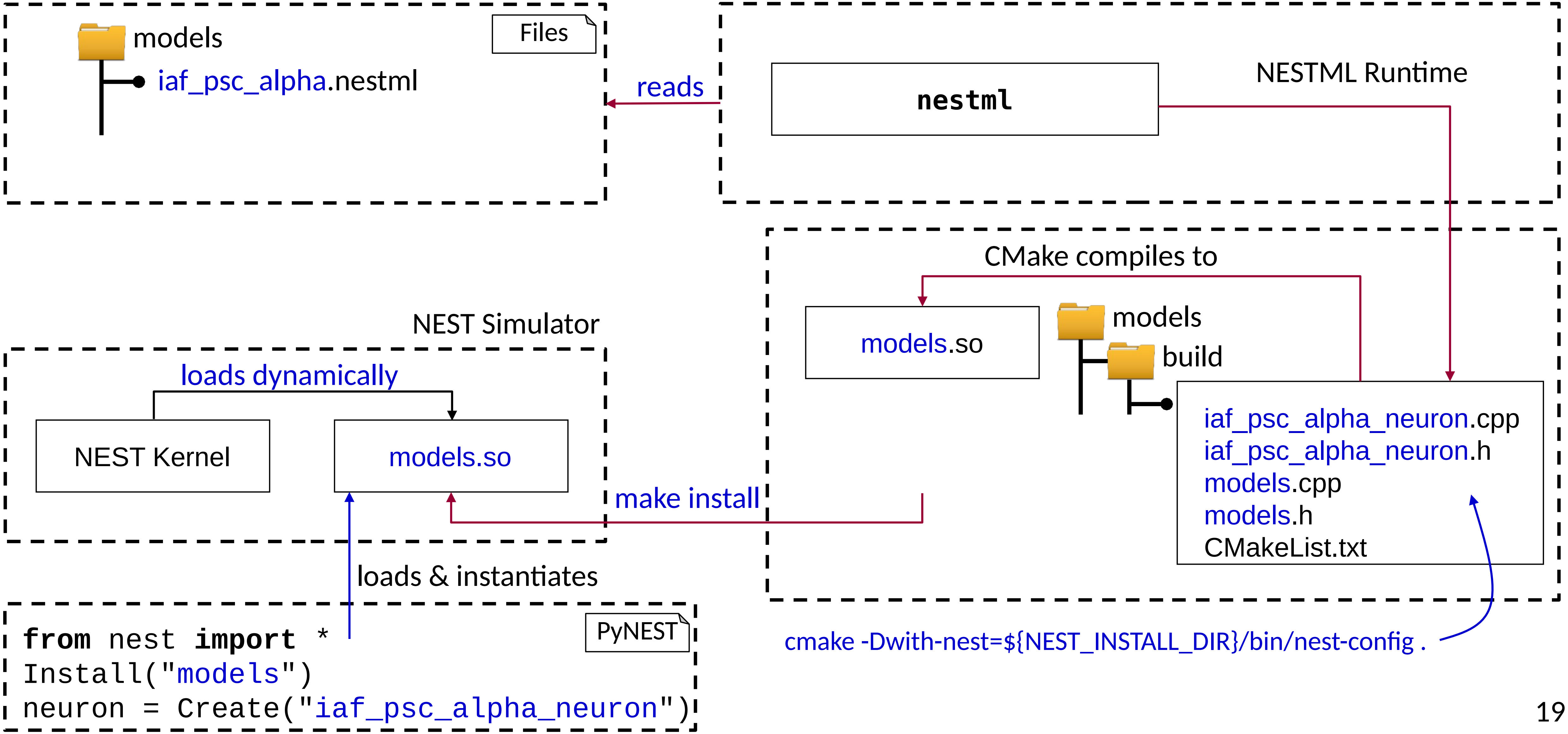
```
neuron rc_currents:
...
  equations:
     I_syn pA = I_e + convolve(I_a, spikes) + currents
     V_m' = -V_m/tau_m + I_syn/C_m
  end

  input:
     currents <- current
     spikes pA <- spike
  end

  output: spike

...

end
```

NEST

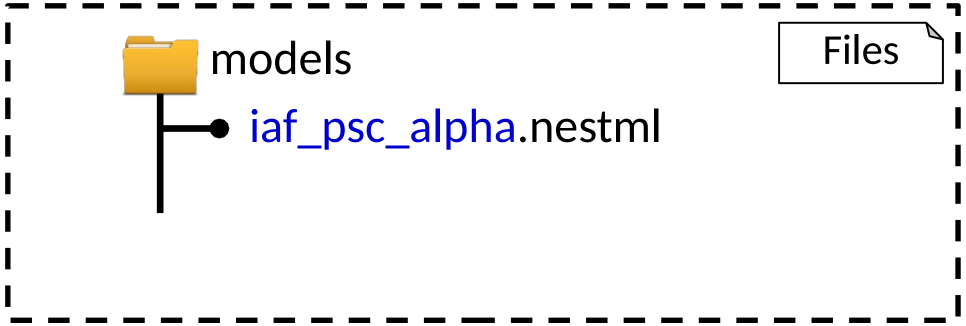
<<Runtime>>



Using NESTML (command line)



Using NESTML (Python)



Make the functions available:

```
from pynestml.frontend.pynestml_frontend import to_nest, install_nest
```

Generate the C++ code:

```
to_nest(input_path="models", target_path="/tmp/module", logging_level="INFO")
```

Compile and install the C++ code:

```
install_nest("/tmp/module", "/home/johndoe/nest-simulator-build")
```

PyNEST API of generated NEST module

PyNEST

```
import nest.*

nest.Install("models")

neuron = Create("rc_neuron")

SetStatus(neuron, {"V_m":-72.0,
                   "C_m":300.0})
mmeter=Create('multimeter')
SetStatus(multimeter1, {"record_from":["V_m"]})

Connect(mmeter, neuron)
```

NESTML

<<rc_neuron.nestml>>

```
neuron rc_neuron:

  initial_values:
    V_m mV = -70mV
  end

  equations:
    V_m' = -(V_m-70mV)/tau_m + I_syn/C_m
  end

  parameters:
    C_m pF   = 250pF
    tau_m ms = 10ms
    I_syn    = 10pA
  end

end
```

Practical exercise: implementing Izhikevich model

- Izhikevich: simple model for spiking neurons
- Work with the `models/izhikevich.nestml` artifact
- State-variables (v, u) are defined through ODEs:

$$v' = 0.04 * v * v + 5 * v + 140 - u + I$$

$$u' = a * (b * v - u)$$

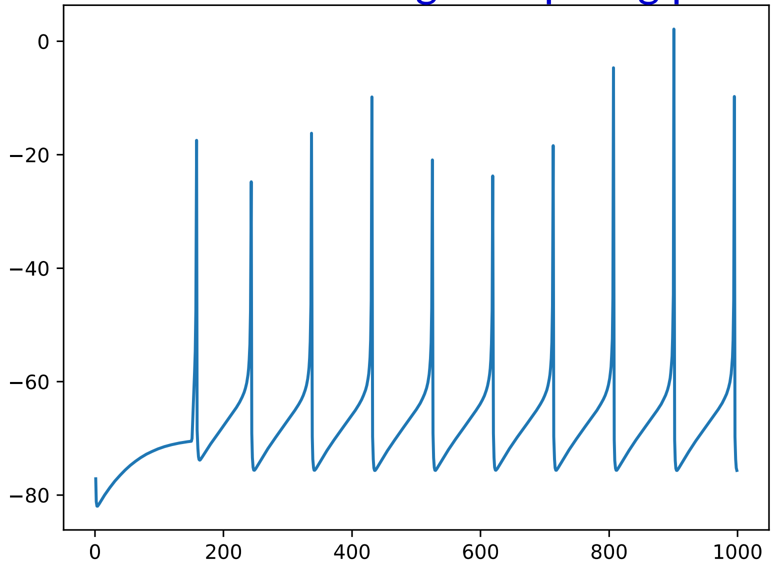
- Parameters (default values for Regular Spiking):

`a=0.02,b=0.2,c=-65.0,d=8.0`

- State-update :

$$\text{if } v \geq 30mV \text{ then } \begin{cases} v = c \\ u = u + d \end{cases}$$

Izhikevich regular spiking plot



Practical exercise: using PyNEST API

- Adjust the run_izhikevich.py script
- Change model's parameter to produce chattering spikes
- Parameters (TODO Chattering):

  `a=0.02,b=0.2,c=-50.0,d=2.0`

- Use the following PyNEST-API
- Change how the neuron is created, e.g.:

```
model_params = {'a': 0.02, ...}
neuron = nest.Create(modelNestml, 1, model_params)
```