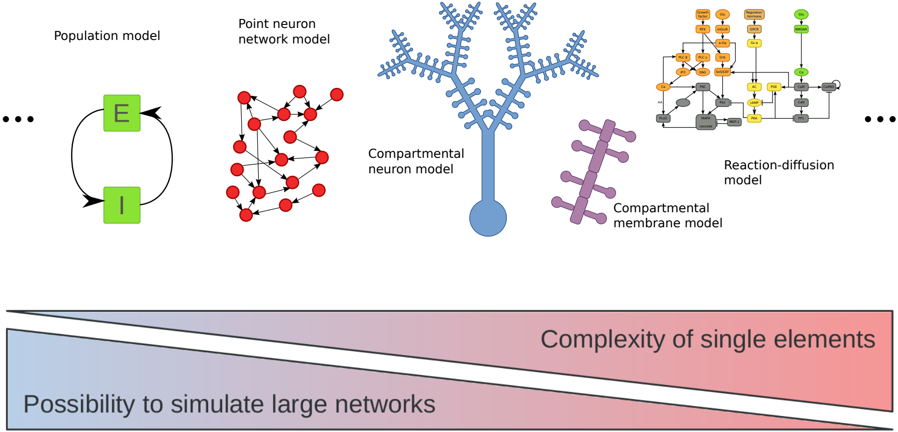
THE NEURAL SIMULATION TOOL NEST

CNS2020 full-day tutorial

July 18th, 2020 | Charl Linssen (c.linssen@fz-juelich.de) | SimLab Neuroscience

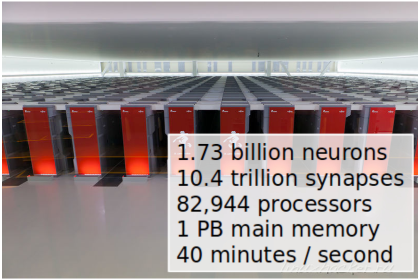
WHEN TO USE NEST?



NEST = NEURAL SIMULATION TOOL

- Point neurons and neurons with few electrical compartments
- Phenomenological synapse models (STDP, STP)
  - + gap junctions, neuromodulation and structural plasticity
- Frameworks for rate models and binary neurons
- Support for neuroscience interfaces (MUSIC, libneurosim)

- Highly efficient C++ core with a Python frontend
- Hybrid parallelization (OpenMP+MPI)
- Same code from laptops to supercomputers



NEST DESIGN GOALS

NEST development is always driven by scientific needs.

High accuracy and flexibility

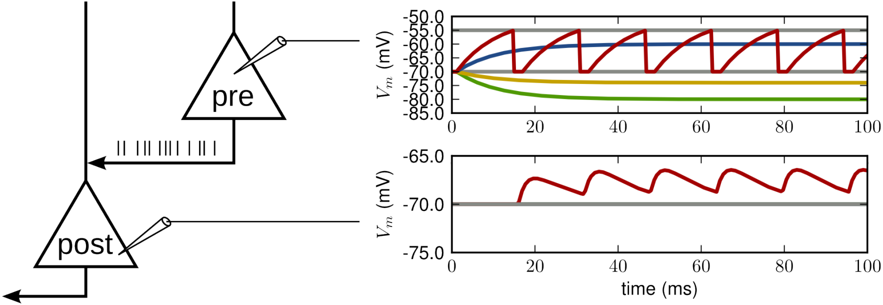
- Exact integration is used for suitable neuron models
- Spike interaction in continuous time available for suitable neuron models
- Extremely scalable: same code from laptop to supercomputers

Constant quality assurance

- Automated unit test suite included in NEST build
- Continuous integration for all repository checkins
- Peer review for all code contributions

NEURONAL SIMULATIONS IN NEST

A simulation in NEST mimics a neuroscientific experiment.



NEURON MODELS

- Integrate-and-fire models (`iaf_`)
  - Current-based (`iaf_psc`)
  - Conductance-based (`iaf_cond`)
  - Different post-synaptic shapes (`_alpha`, `_exp`, `_delta`)
- Single compartment Hodgin-Huxley models (`hh_`)
- Adaptive exponential integrate-and-fire models (`aeif_`)
- Allen Institute generalised leaky integrate-and fire family (`gif_`)
- Hill-Tononi model (Hill & Tononi 2005)
- Neuron models with >1 compartments (“mesocompartmental”)
- . . . and many more!

SYNAPSE MODELS

- Gap junctions (electical synapse)
- Short term depression, facilitation
- Spike-timing dependent plasticity (STDP)
  - All-to-all; several nearest-neighbour variants
- Triplet STDP (Pfister & Gerstner 2006)
- Voltage-based STDP (Clopath et al. 2010)
- Dopamine-modulated STDP (Potjans et al. 2010)
- . . . and many more!

STIMULATION DEVICES

Spike generators:

- `spike_generator` spikes at prescibed points in time
- `poisson_generator` spikes according to a Poisson distribution
- `gamma_sup_generator` spikes according to a Gamma distribution

Current generators

- `ac_generator` provides a sine-shaped current
- `dc_generator` provices a constant current
- `step_current_generator` provides a step-wise constant current
- `noise_generator` provides a random noise current

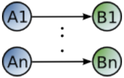
RECORDING DEVICES

- `spike_detector` records incoming spikes
- `multimeter` records analog quantities (potentials, conductances, …)
- `voltmeter` records the membrane potential
- `correlation_detector` records pairwise cross-correlations between the spiking activity of neurons
- `weight_recorder` records the weight of connections

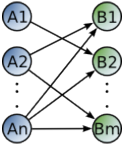
SPECIFICATION OF CONNECTIVITY

- One-to-one
- All-to-all
- Fixed in-, out-degree
- Fixed total number (N)
- Pairwise Bernoulli (p)

```
A = Create('iaf_psc_alpha', n)
B = Create('spike_detector', n)
Connect(A, B, 'one_to_one')
```




```
A = Create("iaf_psc_alpha", n)
B = Create("iaf_psc_alpha", m)
Connect(A, B, {'rule': 'fixed_indegree',
               'indegree': N})
```



JÜLICH Forschungszentrum

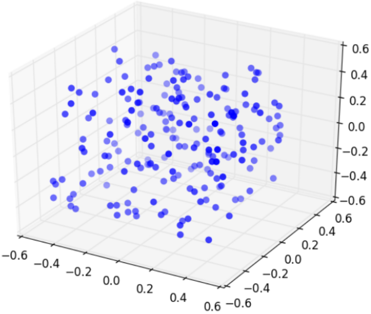
RANDOMIZATION OF SYNAPSE PROPERTIES

Dozens of standard distributions are natively supported.

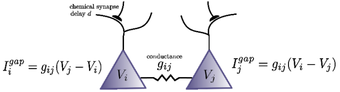
```
delay_dist = {'distribution': 'uniform',
              'low': 0.8, 'high': 2.5}

alpha_dist = {'distribution': 'normal_clipped',
              'low': 0.5, 'mu': 5.0,
              'sigma': 1.0}

syn_dict = {'model': 'stdp_synapse',
            'weight': 2.5,
            'delay': delay_dist,
            'alpha': alpha_dist}
```

STRUCTURED NETWORKS USING TOPOLOGY

- Set node positions on grids or arbitrary points in space (1D, 2D, 3D)
- Nodes can be neurons or combinations of neurons and devices
- Connect nodes in a position- and distance-dependent manner
- Set boundary condition (periodic or not)



GAP JUNCTIONS

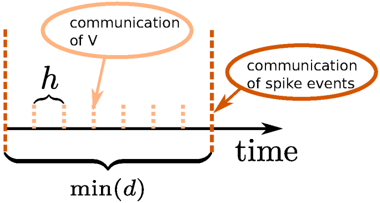


Neuron i (hh_psc_alpha_gap)

$y_i'(t) = f_i(y_i(t))$, $y_i(t_0)$ given

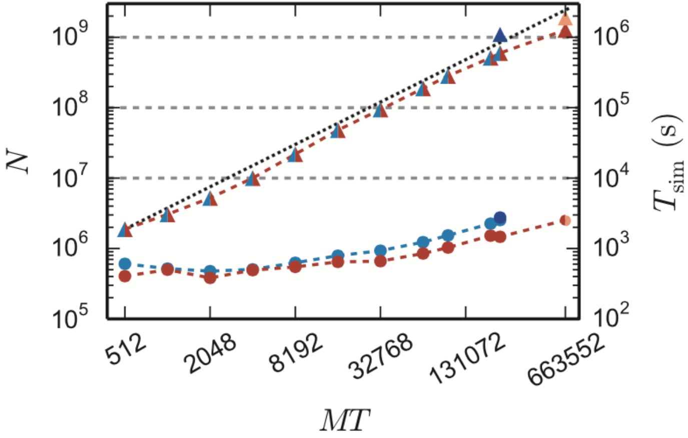
$$\frac{V_i'}{C_m} = -I_i^{ionic}(V_i, m_i, h_i, n_i, p_i)$$
$$+I_i^{applied}(I_i^{ex}, I_i^{in})$$
$$+I_i^{gap}(V_i, V_j)$$

- at each time point neuron i needs membrane potential of neuron j
- large system of differential equations
- naïve: communication of V in each step
- better: Jacobi waveform relaxation



Hahne et al. (2015). A unified framework for spiking and gap-junction interactions in distributed neural network simulations. Frontiers in Neuroinformatics. 9:22

NEST PERFORMANCE



Maximum network size and corresponding run time as function of number of virtual processes on the K computer (red) and JUQUEEN (blue). Taken from Kunkel et al., (2014), Front Neuroinf. DOI: 10.3389/fninf.2014.00078

HOW TO USE NEST?

- The Python interface PyNEST

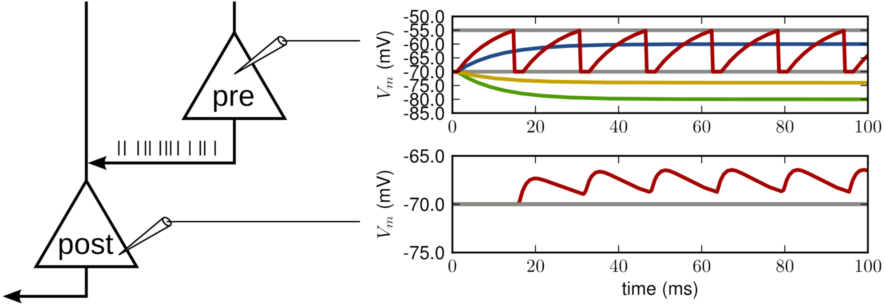
```
import nest
n = nest.Create("iaf_psc_exp", 8000, {"V_m": -65.})
sd = nest.Create("spike_detector")
nest.Connect(n, sd)
...
nest.Simulate(1000.)
```

- The multi-simulator Python interface PyNN

```
from pyNN import nest
nest.setup()
neuron_t = nest.native_cell_type("iaf_psc_exp")
pop_exc = nest.Population(8000, neuron_t, {})
pop_exc.initialize(V_m=-65.)
...
nest.run(1000.)
```

NEURONAL SIMULATIONS IN NEST

A simulation in NEST mimics a neuroscientific experiment.

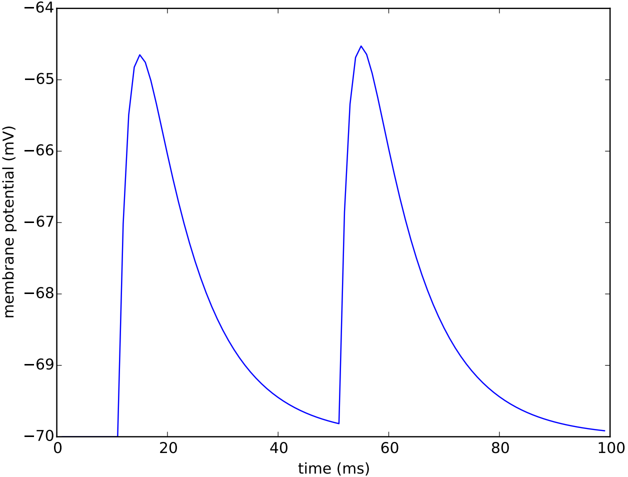


JÜLICH Forschungszentrum

A FULL EXAMPLE

```python
import nest                                        # import NEST module
neuron = nest.Create('iaf_psc_exp')                # create a neuron
voltmeter = nest.Create('voltmeter')               # create a voltmeter
spikegenerator = nest.Create('spike_generator') # create a spike generator
nest.SetStatus(spikegenerator, {'spike_times': [10., 50.]}) # let it spike

# connect spike generator and voltmeter to the neuron
nest.Connect(spikegenerator, neuron, syn_spec={'weight' : 1E3})
nest.Connect(voltmeter, neuron)

nest.Simulate(100.) # run the simulation

# read out recording time and voltage from voltmeter and plot them
times = nest.GetStatus(voltmeter)[0]['events']['times']
voltage = nest.GetStatus(voltmeter)[0]['events']['V_m']
pl.plot(times, voltage)
pl.xlabel('time (ms)'); pl.ylabel('membrane potential (mV)')
pl.show()
```

A FULL EXAMPLE



EVENT-DRIVEN VS. TIME-DRIVEN

Event-driven simulation:

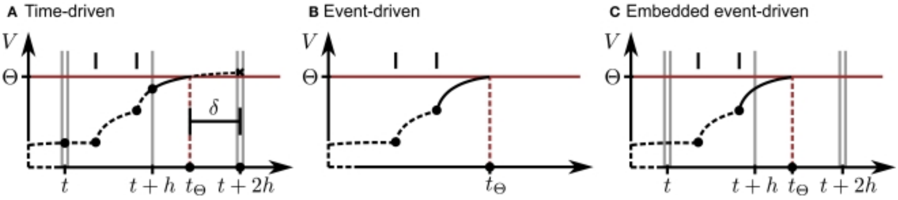
- Visit a neuron only when it receives an event (e.g. a spike)
- From $y(t_i)$, calculate $y(t_{i+1})$

Time-driven simulation:

- Visit each neuron in each time step h
- From $y(ih)$, calculate $y([i+1]h)$

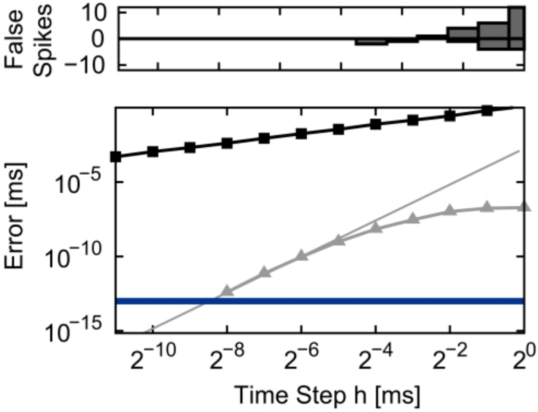
JÜLICH Forschungszentrum

EVENT-DRIVEN VS. TIME-DRIVEN



Hanuschkin et al. Front. Neuroinf. 2010

EVENT-DRIVEN VS. TIME-DRIVEN



Hanuschkin et al. Front. Neuroinf. 2010

EVENT-DRIVEN VS. TIME-DRIVEN

	Event-driven	Time-driven
Pros	■ more efficient for low input rates ■ 'correct' solution for invertible neuron models	■ more efficient for high input rates ■ works for all neuron models ■ scales well
Cons	■ only works for neurons with invertible dynamics ■ event queue does not scale well	■ only 'approximate' solution even for analytically solvable models ■ spikes can be missed due to discrete sampling of membrane potential

EVENT-DRIVEN VS. TIME-DRIVEN

NEST uses a hybrid approach to simulation

- input events to neurons are frequent: time-driven algorithm
  - If the dynamics is nonlinear, we need a numerical method to solve it, e.g.:
    - Forward Euler: $y([i+1]h) = y(ih) + h \cdot \dot{y}(ih)$
    - Runge-Kutta (k-th order)
    - Runge-Kutte-Fehlberg with adaptive step size
    - $\ldots$

  $\rightarrow$ Use a pre-implemented solver, for example, from the GNU Scientific Library (GSL).

  - If the dynamics is linear (e.g. leaky integrate-and-fire), we can solve it exactly.

EVENT-DRIVEN VS. TIME-DRIVEN

NEST uses a hybrid approach to simulation

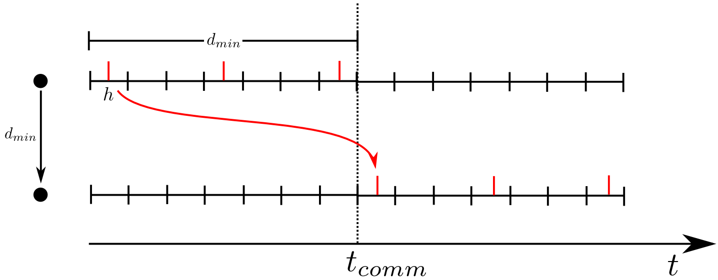
- input events to neurons are frequent: time-driven algorithm
  - If the dynamics is nonlinear, we need a numerical method to solve it, e.g.:
    - Forward Euler: $y([i+1]h) = y(ih) + h \cdot \dot{y}(ih)$
    - Runge-Kutta (k-th order)
    - Runge-Kutte-Fehlberg with adaptive step size
    - ...

  $\rightarrow$ Use a pre-implemented solver, for example, from the GNU Scientific Library (GSL).

  - If the dynamics is linear (e.g. leaky integrate-and-fire), we can solve it exactly.
- events at synapses are rare: event driven component
  - Exception: gap junctions

MINIMUM SYNAPTIC DELAY

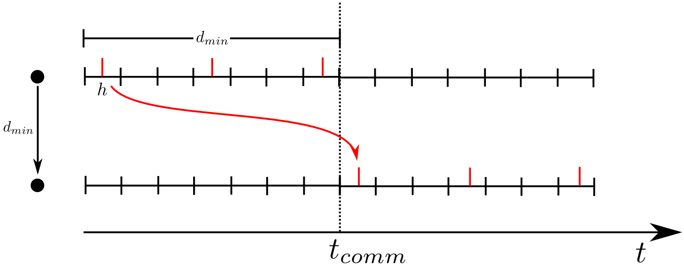
During an interval of the minimal transmission delay in the network (Δ), neurons are effectively decoupled.



Input events to neurons are frequent: time-driven updates.
Events at synapses are rare: event driven.

COMMUNICATION OF EVENTS

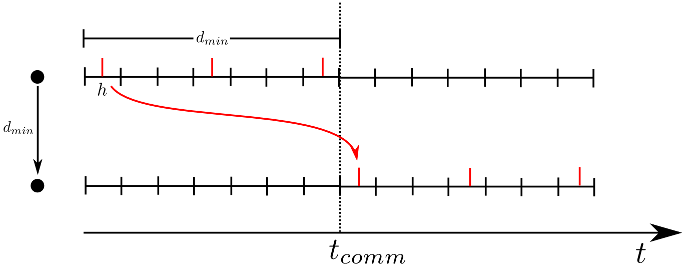
- communication only required in intervals of the minimal delay between neurons



JÜLICH Forschungszentrum

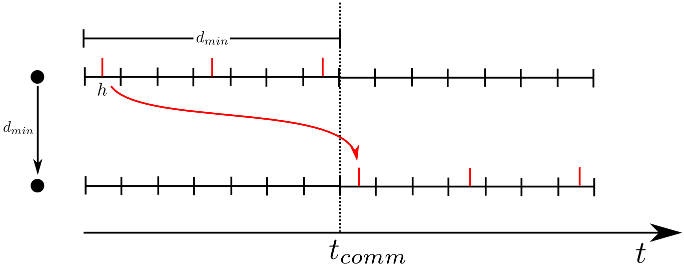
COMMUNICATION OF EVENTS

- communication only required in intervals of the minimal delay between neurons
- communication frequency independent of step size h



COMMUNICATION OF EVENTS

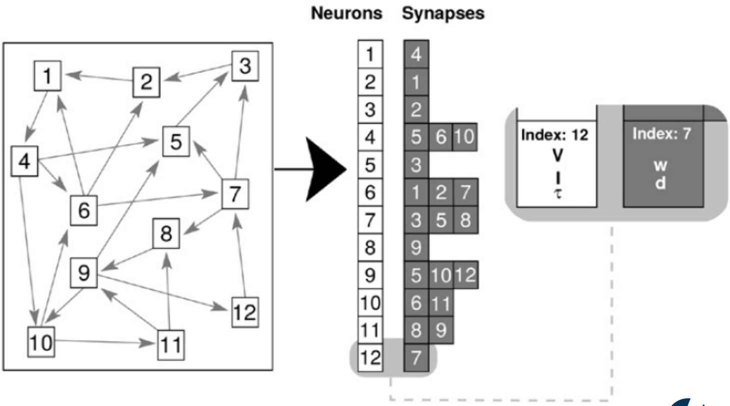
- communication only required in intervals of the minimal delay between neurons
- communication frequency independent of step size h
- less communications containing more data is more efficient due to overhead of communication between machines



JÜLICH Forschungszentrum

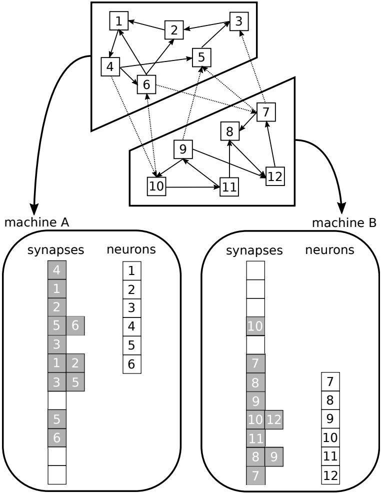
REPRESENTATION OF NETWORK STRUCTURE: SERIAL

- Each neuron and synapse maintains its own parameters
- Synapses save the index of the target neuron



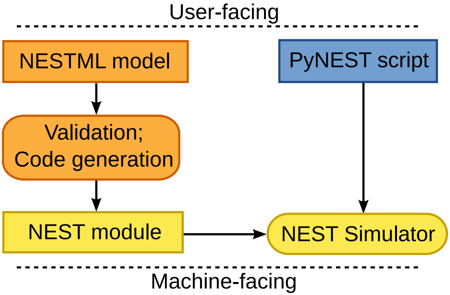
REPRESENTATION OF NETWORK STRUCTURE: DISTRIBUTED

- neurons are distributed round robin onto processes
- one target list for every neuron on each machine
- synapse stored on machine that hosts the target neuron
- wiring is a parallel operation



NESTML

NESTML is a domain-specific language for neuron and synapse models.



Using PyNEST, you instantiate and connect
the models that you define in NESTML.

NESTML: DESIGN PRINCIPLES

- Concise; low on boilerplate
- Speak in the vernacular of the neuroscientist (keywords such as `neuron`, `synapse`)
- Easy (dynamical) equation handling coupled with imperative-style programming (`if V_m >= threshold: ...`)

NESTML comes with a code generation toolbox.

- Code generation (model definition but not instantiation)
- Automated ODE analysis and solver selection
- Flexible addition of targets using Jinja2 templates

NESTML: EXAMPLE

```
neuron iaf_psc_exp:
  state:
    V_abs mV = 0 mV
  end

  equations:
    shape G = exp(-t / tau_syn)
    V_abs' = -V_abs / tau_m
           + (I_ext + convolve(G, spikes)) / C_m
  end

  parameters:
    C_m     pF = 250 pF
    tau_m   ms = 10 ms
    tau_syn ms = 2 ms
    V_threshold mV = 40 mV  # w.r.t. zero!
  end

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

  update:
    integrate_odes()
    if V_abs > V_threshold:
      V_abs = 0
      emit_spike()
    end
  end
end
```

GETTING HELP

Within Python:

```
nest.help('iaf_psc_exp')
nest.help('Connect')
```

Online documentation:

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

Community:

- NEST user mailing list
- Bi-weekly open video conference
- `http://github.com/nest/nest-simulator/`
- Annual NEST Conference: a forum for users and developers

Please tell us about problems. We can only fix what we know of!

REFERENCES AND FURTHER READING

- The NEST Simulator documentation at https://nest-simulator.readthedocs.io/

- Gewaltig et al. (2012) NEST by example: An introduction to the neural simulation tool NEST. doi:10.1007/978-94-007-3858-4_18
- Hanuschkin et al. (2010) A general and efficient method for incorporating precise spike times in globally time-driven simulations. doi:10.3389/fninf.2010.00113
- Jordan et al. (2018) Extremely Scalable Spiking Neuronal Network Simulation Code: From Laptops to Exascale Computers. doi:10.3389/fninf.2018.00002

ACKNOWLEDGMENTS

This presentation is based on previous work by many people.

- Hannah Bos
- David Dahmen
- Moritz Deger
- Jochen Martin Eppler
- Espen Hagen
- Abigail Morrison
- Jannis Schuecker
- Johanna Senk
- Tom Tetzlaff
- Sacha van Albada
- Charl Linssen