frontiers

Draculab: A Python simulator for firing rate neural networks with delayed adaptive connections

Sergio Verduzco-Flores *, **Erik De Schutter**

Computational Neuroscience Unit, Okinawa Institute of Science and Technology

Correspondence*:
Sergio Verduzco-Flores, Computational Neuroscience Unit, Okinawa Institute of Science and Technology,
1919-1 Tancha, Onna-son, Kunigami-gun Okinawa, Japan 904-0495
sergio.verduzco@gmail.com

ABSTRACT

Draculab is a neural simulator with a particular use scenario: firing rate units with delayed connections, using custom-made unit and synapse models, possibly controlling simulated physical plants.

Draculab also has a particular design philosophy. It aims to blur the line between users and developers. Three factors help to achieve this: a simple design using Python's data structures, extensive use of standard libraries, and profusely commented source code.

This paper is an introduction to Draculab's architecture and philosophy. After presenting some example networks it explains basic algorithms and data structures that constitute the essence of this approach. The relation with other simulators is discussed, as well as the reasons why connection delays and interaction with simulated physical systems are emphasized.

Keywords: neural simulator, firing rate units, Python, transmission delays, adaptive synapses

1 INTRODUCTION

When faced with a new project, modelers in computational neuroscience must decide whether to program their simulations from scratch, or to use one of the existing neural simulators. Draculab was born from a project whose requirements were not met by any other simulator. Those requirements were:

- Firing rate units that operate as continuous-time dynamical systems, connected with transmission delays.
- Experimental types of units and synapses, with frequent modifications happening.
- Simulations where neural controllers interact with a physical plant, forming a closed-loop system.

Besides satisfying these requirements, Draculab aims to have a simple interface, but still provide total control over the simulation to experienced Python users. For users with basic command of Python, or in a hurry to simulate, Draculab is a neural simulator with an interface similar to PyNEST Eppler et al. (2009). For more experienced users, it is an intuitive solution to the task of simulating ordinary differential equations (ODEs) connected through delay lines; a solution that can be adapted to their own needs.

Following Brian's Goodman and Brette (2009) insight that the main limiting factor is not always computational efficiency, Draculab is written entirely in Python, with certain key functions coded in Cython Behnel et al. (2011) (`https://cython.org/`) for speed. Development environments such as Spyder (`https://www.spyder-ide.org/`), or the Jupyter Notebook (`https://jupyter.org/`), are well suited for working with Draculab. The interface uses standard functions that create units, connect them, and run simulations. These functions are configured using parameter dictionaries as their arguments. Users can launch their first simulations within minutes (see section 2).

It is expected that people who want to write a simulation don't want to learn a new programming language, or the intricacies of a complex interface. On the other hand, some users want to write very non-standard simulations, using their own synaptic plasticity rules, or their own firing rate models, perhaps connecting them with other dynamical systems. The idea to resolve this is to take advantage of the users' knowledge of Python. People who already understand Python don't have to learn many new things, except for the basic architecture of the system, which is described in the sections below.

For experienced Python users, Draculab aims to give ample control over the simulation, so it feels like they wrote the simulator themselves (but without going through the hassle of writing and testing it). This comes from a straightforward implementation, profusely commented source code, and standard libraries such as Scipy (`https://www.scipy.org/`), Numpy (`https://www.numpy.org`), and Matplotlib (`https://matplotlib.org/` for numerical routines and visualization.

Section 2 introduces some examples so the reader can become acquainted with Draculab's API, and some use scenarios. The sections thereafter provide an introduction to Draculab and its architecture in a way that the reader can gain considerable understanding at the source code level, but without introducing excessive detail. Readers who want to proceed further can work through the tutorials, or consult the docstring documentation.

2 EXAMPLE NETWORKS

This section presents the source code of two Draculab simulations.

2.1 A first example

To begin with, let's create a network with 10 sigmoidal units and one input. It does nothing special, but it illustrates the basic interface of the simulator.

We segment the source code of this example code into 5 steps, and analyze each one. The full source code can be seen in Appendix A.

1. CREATE A NETWORK OBJECT

```
net_params = {'min_delay': 0.1, # minimum connection delay
              'min_buff_size': 10 } # minimum buffer size
net = network(net_params) # creates the network
```

The basic object used to run Draculab simulations is an instance of the class `network`. The constructor of this class requires a parameter dictionary with the smallest delay among all the connections (`min_delay`), and the number of values to be stored by each unit in the timespan

of the smallest delay (`min_buff_size`). The significance of these parameters will be explained in section 3.

2. POPULATE THE NETWORK WITH TWO TYPES OF UNITS

```
sig_params = {'type': unit_types.sigmoidal, # unit model
              'init_val': 0.5, # initial value for all sigmoidal units
              'slope': 1,   # slope of the sigmoidal function
              'thresh': 0., # threshold of the sigmoidal function
              'tau': 0.2,   # time constant of the sigmoidal unit
              'tau_fast': 0.1 } # time constant of the 'fast' low-pass filter
inp_params = {'type': unit_types.source, # source units provide inputs
              'init_val': 1.,
              'function': lambda t: np.cos(t) } # a cosine function
sig_units = net.create(10, sig_params) # create 10 sigmoidal units
inp_unit = net.create(1, inp_params) # create 1 input unit
```

First we create parameter dictionaries that configure the sigmoidal and the input units. All units have `type` and `init_val` attributes; other parameters to include depend on the type of the unit. The specifics can be consulted in the unit's documentation. For example, the Python command `help(sigmoidal.__init__)` will show the entries expected in the parameter dictionary for sigmoidal units.

Usually units produce an output based on their inputs and their current state. `source` units are different, since their output comes from a Python function defined by the user, whose argument is the simulation time. This is a flexible way to specify inputs. Source units can also be used to report the value of any variable in the simulation (such as the synaptic weights) as it evolves through time.

Upon creation, each unit is assigned an integer that uniquely identifies it. The `create` method returns a list with the identifiers of the created units.

3. CONNECT THE UNITS

```
# create connections between the sigmoidal units
conn_spec = {'rule': 'fixed_outdegree', # rule to create connections
             'outdegree': 2, # each unit sends 2 projections randomly
             'delay': 0.2 } # all connections have a delay of .2 time units
syn_spec = {'type': synapse_types.oja, # synapses use the Oja learning rule
            'init_w': {'distribution': 'uniform', 'low': 0.1, 'high': 1.},
            'lrate': 0.1 } # learning rate for the Oja rule
net.connect(sig_units, sig_units, conn_spec, syn_spec) # create the connections
# connect the input to the sigmoidal units
conn_spec_2 = {'rule': 'all_to_all', # rule to create connections
               'delay': 0.1 } # all connections have a delay of .1 time units
syn_spec_2 = {'type': synapse_types.static, # synapses don't change
```

```
            'init_w': 0.5 } # all synapses have this initial weight
net.connect(inp_unit, sig_units, conn_spec_2, syn_spec_2) # create the connectio
```

Users familiar with PyNEST may find many similarities in how the `connect` method works. PyNEST was used as a template on how to specify connections, and Draculab has a `topology` module that resembles the one in PyNEST, although they are not identical (see section 7). In the example code, we create parameter dictionaries to configure the connections, and then call the `connect` method to create them. Running the Python instruction `help(network.connect)` can yield more details.

In the `syn_spec` dictionary it can be observed that the synapses used for connections between sigmoidal units are of the `oja` type. The *Oja* learning rule Oja (1982) is a Hebbian-type model, usually associated with principal component extraction. In Draculab a continuous-time version of this model is implemented in the `oja_synapse` object. The `network.connect` function creates a synapse object for each connection. The name of all implemented synapse models can be obtained by typing the command `synapse_types.list_names()`.

4. RUN THE SIMULATION

```
# run the simulation for 10 time units
sim_data = net.run(10.)
```

The `run` method receives a time to simulate, and returns a tuple containing simulation data. Calling `run` repeatedly restarts the simulation from its last time point. It should be noted that in Draculab no particular units of time, distance, or firing rate are enforced.

5. PLOT THE RESULTS

```
# visualize using Matplotlib
sig_activs = np.array(sim_data[1])[sig_units]
plt.plot(sim_data[0], sig_activs.transpose())
plt.show()
```

In here we use the `plot` function from Matplotlib to visualize the activities of the 10 sigmoidal units. The resulting plot is similar to the one presented in Figure 1.

2.2 A closed-loop simulation

Next is one of the simplest networks that perform feedback control of a physical plant, with synaptic plasticity enhancing performance. The plant being controlled is a pendulum, and a single linear *control* unit acts as a proportional feedback controller that puts the pendulum at a desired angle. The control unit receives an error signal from a source unit, and the state of the pendulum from four *afferent* sigmoidal units. Learning takes place at the synapses from the afferent units, using the *input correlation* rule Porr and Wörgötter (2006). The connectivity of this network can be observed in figure 2.

The input correlation rule is defined by the following equation:

$$\frac{d\omega_j}{dt} = \mu u_j \frac{du_0}{dt},$$

where ω_j is the weight of the input from the j-th unit, which has activity u_j. Unit 0 provides an error signal, and by using its derivative the input correlation rule seeks to exploit correlations between input signals and error increase. This is implemented as the `inp_corr` synapse type. Further details about this network can be found in Draculab's tutorial.

First we create the `ic_control` class, which contains the procedures used to run a simulation. The idea is that after creating an instance of `ic_control` we can modify its parameter dictionaries, and then run `ic_control.initialize` to create a network with the desired modifications. Simulations are then performed using `ic_control.simulate`.

```
class ic_control():
   def __init__(self):
   ########### Create parameter dictionaries for network, plant, and units
      # parameter dictionary for the network
      self.net_params = {'min_delay' : 0.01,
                  'min_buff_size' : 10,
                  'rtol' : 1e-5,
                  'atol' : 1e-5 }
      # This pendulum has no gravity (horizontal pendulum).
      # Initial position is 45 degrees.
      self.plant_params = {'type' : plant_models.pendulum,
                    'length' : 2., # length in meters
                    'inp_gain' : 10., # input gain
                    'g' : 0.,      # gravitation constant
                    'mass' : 10.,  # in kilograms
                    'mu' : 1.,     # viscous friction coefficient
                    'init_angle' : np.pi/2., # initial angle in radians
                    'init_ang_vel' : -0.2} # initial angular velocity
      # parameter dictionary for the afferent input units
      self.aff_unit_pars = {'init_val' : 0.5,
                     'type' : unit_types.sigmoidal,
                     'tau_fast' : 0.01,
                     'slope' : [0.5, 0.5, 0.2, 0.2],
                     'thresh' : [0.2, 0.2, 0.2, 0.2],
                     'tau' : 0.02}
      # parameter dictionary for the "controller" unit using input correlation
      self.ic_unit_pars = {'init_val' : 0.5,
                    'type' : unit_types.linear,
                    'tau' : 0.02}
      # parameter dictionary for the source unit providing
      # the error signal to the ic unit
      self.err_unit_pars = {'init_val' : 0.,
```

```
                        'tau_fast' : 0.01,
                        'tau_mid' : 0.1,
                        'function' : lambda x:None,
                        'type' : unit_types.source}
        ########### Create parameter dictionaries for the synapses and connections
            self.pend2affs_syn_spec = {'init_w' : [1., -1., 1., -1.],
                                'type' : synapse_types.static}
            self.ic2pend_syn_spec = {'init_w' : 1., 'type' : synapse_types.static}
            self.affs2ic_syn_spec = {'init_w' : 0., 'lrate' : 15.,
                            'input_type' : 'pred', 'type' : synapse_types.inp_corr}
            self.err2ic_syn_spec = {'init_w' : 2., 'lrate' : 0.1,
                           'input_type' : 'error', 'type' : synapse_types.inp_corr}

            # port map: first two units receive angle,
            # second two receive angular velocity
            self.pend2affs_conn_spec = {
                            'port_map' : [ [(0,0)], [(0,0)], [(1,0)], [(1,0)] ],
                            'delays' : 0.01}
            self.ic2pend_conn_spec = {'inp_ports' : [0], 'delays' : 0.02}
            self.affs2ic_conn_spec = {'rule' : 'all_to_all', 'delay' : 0.02}
            self.err2ic_conn_spec = {'rule' : 'all_to_all', 'delay' : 0.02}

        def ang_dev(self, t):
        # using this as the error, the ic
        # effectively turns into a proportional controller
            pend_ang = self.net.plants[self.pend].get_angle(t) # angle modulo 2pi
            # Smallest rotation required to go from pend_ang to des_ang:
            return (self.des_ang - pend_ang + np.pi) % (2*np.pi) - np.pi

        def initialize(self):
        ########### Create network, plant, and input units
            self.net = network(self.net_params) # creating a network
            self.pend = self.net.create(1, self.plant_params) # a pendulum
            self.affs = self.net.create(4, self.aff_unit_pars) # afferent units
            self.ic = self.net.create(1, self.ic_unit_pars) # controller unit
            self.err = self.net.create(1, self.err_unit_pars) # error unit

        ########### Set the error function
            self.des_ang = 0. # desired angle in range [0,2*pi)
            self.net.units[self.err[0]].set_function( self.ang_dev )

        ########### Connect
            # afferents to controller
            self.net.connect(self.affs, self.ic, self.affs2ic_conn_spec,
                        self.affs2ic_syn_spec)
```

```
        # error to controller
        self.net.connect(self.err, self.ic, self.err2ic_conn_spec,
                    self.err2ic_syn_spec)
        # controller to plant
        self.net.set_plant_inputs(self.ic, self.pend, self.ic2pend_conn_spec,
                          self.ic2pend_syn_spec)
        # plant to afferents
        self.net.set_plant_outputs(self.pend, self.affs, self.pend2affs_conn_spec,
                           self.pend2affs_syn_spec)

    ########## To keep track of error 'derivative', use an isolated source unit
        track_unit_pars = {'init_val' : 0., 'function' : lambda x:None,
                     'type' : unit_types.source}
        self.track = self.net.create(1, track_unit_pars)
        self.net.units[self.track[0]].set_function(
                           lambda x : self.net.units[self.ic[0]].err_diff )

    def simulate(self, sim_time):
    ########### Simulate
        start_time = time.time()
        self.times, self.unit_stor, self.plant_stor = self.net.run(sim_time)
        print('Execution time: %s seconds' % (time.time() - start_time))

    def plot_results(self):
    #---- OMITTED FOR BREVITY ----#
```

Both units and plants may receive qualitatively different types of inputs. For example, units may have inputs that have a modulatory, rather than excitatory effect. Plants may have separate inputs that act on different state variables, such as torques on different joints of a double pendulum. To handle this Draculab uses *input ports*, which is a concept inspired by the NEST simulator Diesmann and Gewaltig (2001). The `syn_spec` parameter dictionary of the `network.connect` method can specify an input port, which gets stored in the synapse object. How different ports are interpreted depends on the particular unit and plant models, opening many possibilities, such as inputs that target distinct regions of the dendritic tree (e.g. London and Häusser (2005)).

In the code above, specification of which plant port receives a unit's output is in the `inp_ports` attribute of the `ic2pend_conn_spec` parameter dictionary, which is an argument to the `set_plant_inputs` method. In a similar manner, there is a `set_plant_outputs` method that receives a parameter called `port_map` specifying which plant outputs are connected to which of the units' ports. Details can be read in the documentation of the relevant methods.

At this point we can compare the performance of the network with and without the input correlation rule. First we set the learning rate to zero so the control unit becomes a proportional controller, only driven by the error:

```
""" Simulate using no IC learning """
```

```
icc = ic_control()
icc.affs2ic_syn_spec['lrate'] = 0. # learning rate of input correlation rule
icc.initialize()
icc.simulate(30.)
icc.plot_results()
```

The results of this simulation can be seen in figure 3. It is clear that the system is not stable, as would be expected from a proportional controller in an underdamped system. Next we simulate with a non-zero learning rate for the input correlation rule.

```
""" Simulate using no IC learning """
icc = ic_control()
icc.affs2ic_syn_spec['lrate'] = 40. # learning rate of input correlation rule
icc.initialize()
icc.simulate(30.)
icc.plot_results()
```

The results can be seen in figure 4. The input correlation rule is surprisingly effective, despite not using input filter banks Porr and Wörgötter (2006).

3 THE DRACULAB APPROACH TO NEURAL SIMULATION

A Draculab network is a collection of interconnected units, optionally interacting with a plant.

Units are continuous-time dynamical systems that implement models of firing-rate neurons with one-dimensional outputs. Connections between units have a temporal delay and a synapse type. Temporal delays are multiples of a minimum delay. Synapses can too be dynamical systems, modifying their weights in response to presynaptic and postsynaptic events.

Plants are continuous dynamical systems that can be modelled with ordinary differential equations. Like units, they can receive input connections from multiple units, and unlike units they can produce several output values, since they can have any finite dimensionality. Plants can be used to model a physical system that is being controlled by the units in the network.

A standard approach used to numerically simulate a system of ODEs with delay connections is to have a minimum delay (called `min_delay`). If you store past states of the system in a set of buffers, then it is possible to advance state of the system by `min_delay` time units.

For example, in the system:

$$\frac{x(t)}{dt} = y(t-0.2), \quad \frac{y(t)}{dt} = -x(t-0.5);$$

there is a minimum delay of 0.2 seconds, and to simulate the system we need to store values of x spanning the last 0.5 seconds, and values of y spanning the last 0.2 seconds. Using this we can advance the state from time t to time $t + 0.2$ using a standard ODE solver. The simulation can thus proceed with time steps of size 0.2, although the numerical solver may calculate many states at intermediate time points for each time step.

Accordingly, in Draculab every unit has a buffer with past activation values spanning a time interval equal to the longest delay in the unit's projections. The network parameter `min_buff_size` in the first example of section 2 indicates how many values are stored for a `min_delay` time period in all buffers; the number of past activation values in a unit's buffer depends on how many `min_delay` time periods it spans. If unit A sends a projection to unit B, when B is updating its state it will request unit A for any past values that it requires. Unit A will respond to these requests by using linear interpolation on the values stored in its buffer. For some implementations of ODE integration methods with fixed step size linear interpolation may not be required (see section 5).

In Draculab this is transparent to a user writing a new unit model. All the user needs to do is to create a new class for their custom unit, and to have that new class inherit from the `unit` class. This will provide the new class with the methods that handle buffers and integration, which use Cython optimizations and Scipy's `odeint` numerical solver by default; a different solver can be specified for a unit with the `integ_meth` attribute of its parameter dictionary. The new unit class only needs to specify a constructor (`__init__`) to initialize its variables, and a `derivatives(y, t)` method, that will tell the numerical ODE solver what's the derivative of the unit's firing rate at time t, given that the current state is y. Appendix B provides the code implementing a simple linear unit, where this can be seen explicitly. All the functionality of the unit is implemented using 3 lines.

At this point it is also possible to read the method used to run the simulations (`network.run`), whose source code is in Appendix C. The reader is encouraged to have a look at this short example.

A unit has a single output, but it may many more dynamical variables used to assist the computation of synaptic dynamics and its own firing rate. The firing rate of the unit is a *fast* variable, integrated with the `odeint` solver. Units may have other variables, often evolving on slower timescales, updated every `min_delay` time units with a user defined *update* method. These variables are usually required to compute synaptic plasticity. For example, in the Law and Cooper version of the BCM plasticity rule Law and Cooper (1994) the synaptic weights evolve according to the equation:

$$w' = \alpha x_{pre} x_{post} \left(x_{post} - \theta\right) / \theta$$

where x_{pre} and x_{post} are the presynaptic and postsynaptic firing rates, α is the learning rate, and θ is the average of the squared postsynaptic activity. To avoid duplicate calculations, a variable like θ should be computed by the units rather than by the synapse objects. In this case, θ can be obtained using a first order low-pass filter on x_{post}, and when this is implemented the unit in effect updates two state variables: its firing rate, and the low-pass filtered value, which is a synaptic *requirement*.

"Requirements" are updated every time the `unit.update` function is called. This is also the function that updates the firing rate and the unit's buffer. `unit.update` handles requirements by calling the function `pre_syn_update`, which invokes all functions required for this end. If the user wants to write a plasticity rule that uses a value such as the sum of inputs, or something more exotic, this can be done by writing a function that updates the required value, and by including that function among the ones called by `pre_syn_update`.

A design convention in Draculab is that synapses don't have buffers to store past activation values, as units do. Synapses are updated once per simulation step, and as far as the unit's firing rate dynamics are concerned, the synaptic weights are constant during this `min_delay` period. This comes from the observation that the changes in synaptic weights tend to be much slower than the changes in firing rate, so we might as well update them using a simple integration rule (such as forward Euler) explicitly

written in the synapse's update method. This can bring significant gains in efficiency. For example, the `bcm_synapse` class implements the BCM learning rule described above, and has the following `update` method, getting called once per simulation step by the `unit.update` method of its postsynapic unit.

```
    def update(self, time):
        """ Update the weight using the BCM rule. """
        post = self.net.units[self.postID].get_lpf_fast(0)
        avg_sq = self.net.units[self.postID].sq_lpf_slow
        pre = self.net.units[self.preID].get_lpf_fast(self.delay_steps)
        # A forward Euler step
        self.w = self.w + self.alpha * post * (post - avg_sq) * pre / avg_sq
```

There are some clarifications to make about this piece of code:

- `self` refers to the synapse object, `self.net` to the network object, and `sel.net.units` is a list that contains all the units in the network. The index of a unit in this list is its unique identifier (`ID`). The synapse keeps the identifiers of its postsynaptic unit (`postID`), and its presynaptic unit (`postID`). The value `alpha` is the length of the simulation step times a learning constant.
- This implementation of the learning rule does not really use the pre- and postsynapic firing rates. Instead it applies low-pass filters with a fast time constant to the firing rates, and uses that in the equations. A version of the firing rate passed through a first order low-pass filter is a standard synaptic requirement in Draculab, as it can promote stability in continuous-time sytems. Draculab uses the analytical solution of the filter's differential equation in order to implement an explicit solver. A unit's `lpf_fast` value is its "fast" low-pass filtered activity, and a unit's `get_lpf_fast(n)` method returns the `lpf_fast` value as it was `n` simulation steps before (to account for transmission delays).

Plants are very similar to units, but they may have more than one state variable that gets updated by the `odeint` solver. Units can send projections to plants, and plants can send projections to units, in both case mediated by synapses. Direct connections between plants is not supported.

As with units, to create a new plant a user has to define a class. This class only requires `__init__` and `derivatives` methods to be defined.

From the information so far it is possible to get a general outline of how a Draculab simulation proceeds when `network.run` (Appendix C) is executed:

1. The network object stores all unit and plant values from the previous step.
2. The network object requests all units and plants update their state.
3. In `unit.update`:
   a. Units first update the content of their buffers from the current simulation time `t`, to `t+min_delay`, using the values in the buffers of all other units, and their own integration method (`odeint` by default).
   b. Units run `pre_syn_update`, which updates the values of all their requirements.
   c. Units call the `update` method of all synapses providing them inputs. This updates their weights.
4. In `plant.update` the plants update the content of their buffers.

The next section brings a little more detail to this process.

4 THE ESSENTIAL DRACULAB DATA STRUCTURES

Draculab uses four basic classes to run simulations: `network`, `unit`, `synapse`, and `plant`. These comprise the core simulator. Each instance of the `network` class is a full neural network with its own set of units, synapses, and plants, which the network stores in 3 lists named `units`, `syns`, and `plants`. As mentioned in section 3, each unit has an identifier `ID`, which is its index in the `units` list.

Representing the connectivity information in a network with no delays can be done with a weight matrix. Furthermore, the product of the weight matrix times the vector of unit activities conveniently provides the sum of inputs times their synaptic weights for all units. This is not so straightforward when there are connection delays, so a new approach is needed. Draculab provides two different options to represent connections. The first one, to be described in the next section, uses advanced Numpy array indexing so that each unit has an index representing all its inputs and their delays. The second one, to be described in this section, solves all connectivity issues using 3 lists: the aforementioned `syns`, and two other lists called `delays` and `act`.

Each element in `syns` is another list. The list `syns[i]` contains all the synapses from the projections received by the unit with `ID=i`. `syns[i][j]` is the synapse object for the `j`-th connection to the `i`-th unit. The `delays` list has the same structure: `delays[i][j]` is the delay of the `j`-th connection to the `i`-th unit. `act` has also this structure, but it requires some clarifications.

Every unit has a `get_act(time)` method that provides its activity (firing rate) as a function of time. This method obtains the activity using interpolation on the values of the unit's buffer. The value `act[i][j]` contains a reference to the `get_act` method of the presynaptic unit for the `j`-th connection to unit `i`.

Using these three lists it is simple to get the `j`-th input to unit `i` at time `t`: it is `act[i][j](t - delays[i][j])`. To further illustrate this, here is a plausible implementation of the `unit.get_input_sum` method, which provides the sum of inputs times their synaptic weights:

```
sums = 0.
for j in range(len(syns[ID])):
   sums += syns[ID][j].w * act[ID][j](t - delays[ID][j])
return sums
```

The actual implementation uses a list comprehension, and may not be as readable to some, but it gets everything done in a single line:

```
def get_input_sum(self,time):
   """ Returns the sum of all inputs at the given time,
      each scaled by its synaptic weight.
      The sum accounts for transmission delays. Input ports are ignored.
   """
   return sum([ syn.w * fun(time-dely) for syn,fun,dely in
            zip(self.net.syns[self.ID],
               self.net.act[self.ID],
               self.net.delays[self.ID]) ])
```

Lists can offer a clear solution to the problem of connectivity. It is not the fastest way, but it agrees with the principles of Python and Draculab. Simple is better than complex. Still, sometimes we may want to trade some simplicity for the sake of speed, so the approach of section 5 is offered.

5 IN SEARCH FOR SPEED

The architecture described so far was created with simplicity and flexibility in mind, but it would be great if it could be fast too. Speed, however, is limited because the `get_input_sum` method shown above relies on Python data structures. In particular, the connectivity structure is described by the `delays` and `act` nested lists. The speed bottleneck imposed by this can be broken through a design we call the *flat network*.

In a flat network the data of all the unit buffers is placed in a single 2-dimensional numpy array in the `network` class, called `acts`. It could be said that all the unit buffers are *flattened* into `acts`. The unit class still retains its `buffer` attribute, but it now becomes a *view* of a slice of the `acts` array. Being a view means that the buffer of a unit is a numpy array that uses the same memory addresses as its corresponding entries in the `acts` array.

With the past activation data for all units contained in the `acts` array, connectivity and delays can be represented by having a structured index in each unit, so that when applied to `acts`, it retrieves all the input values that the unit needs to calculate its input sum at several time points. In other words, if a unit has an `idx` index, then `acts[idx]` will return an array with all the inputs the unit received during a `min_delay` time period. In the current implementation, if the unit has m inputs, then `acts[idx]` has m rows and `min_buff_size` columns. Using this array is straightforward to calculate the input sums to the unit.

To the user all of this is transparent. Creating a flat network is identical to creating a normal Draculab network. The difference is that the `run` method used to start simulations is substituted by the `flat_run` method, which automatically uses the flat network. The first thing that `flat_run` does is to call the `network.flatten` method, which moves the unit buffers into the `acts` array, along with other preparations. The rest of `flat_run` is very similar to `run` except for the method that updates the units' activity.

To a developer writing a Draculab unit model, there is a small difference between writing it for a regular or a flat network. As mentioned before, for the regular network the user needs to write `init` and `derivatives` methods. For a flat network a `dt_fun` method must be written instead of `derivatives`. The difference between these two is that whereas `derivatives` retrieves the input sum using the `get_input_sum` method, `dt_fun` retrieves the input sum directly from an array in the unit called `inp_sum`. For example, here is the `dt_fun` method for linear units:

```
def dt_fun(self, y, s):
    """ The derivatives function used when the network is flat. """
    return ( self.inp_sum[s] - y ) * self.rtau
```

As this suggests, for most developers using a flat network will not bring an increase in complexity. Flat networks are more challenging only for those wanting to write new integration methods for the unit, as this entails advanced array indexing. Currently, all integration methods used for flat networks use a fixed step size, because with these it is not necessary to interpolate in order to obtain the required

past activation values. There are versions of the "Euler", "Euler-Maruyama", and "Exponential Euler" integration methods using this scheme.

The reduction in execution time from using a flat, non-interpolating integration method is dependent on the specific structure of the network, but it is common to observe that a flat network is at least 3 times faster than the analogous regular network.

Readers who have progressed this far have already peeked at the heart of Draculab, and should have no major trouble in creating custom models and tweaking things after going through the tutorials.

6 NOISE

Noise can be an important component of neural computations Swain and Longtin (2006); Destexhe and Contreras (2006), and is sometimes part of rate models Hahne et al. (2017).

External noise present in the inputs to a unit is straightforward to implement using source units. A source unit that provides noise input can be created by assigning it a function that produces random values from a particular distribution. It is recommended to use the `numpy.random` library for this purpose, because Draculab uses this library for its own random routines (e.g. to create random initial weights or random connectivity). Thus, if `numpy.random` is used to provide noisy inputs then a single seed initialization (e.g. an instruction like `numpy.random.seed(546789054)`) permits to reproduce the simulation with the same random values.

Intrinsic noise can also be added to the unit models, turning their ODEs into Langevin equations. Specifically, if the equation of the model is:

$$x'(t) = f(x, t)$$

it can be turned into the stochastic differential equation:

$$dx(t) = f(x, t)dt + \sigma dW(t)$$

where $W(t)$ denotes a Wiener process with unit variance.

Turning the ODE into an SDE can be done by substituting the `odeint` numerical solver by a stochastic integrator. This can be done by specifying a different integration method with the `integ_meth` entry of the unit's parameter dictionary. Currently two stochastic integrators (Euler-Maruyama and "stochastic exponential Euler") are available, as Cython utilities for regular networks, and as methods of the `unit` class for flat networks.

It is easy to create simulations where different units have different integration methods. The one restriction is that all integration methods must be either for regular, or for flat networks. The integration methods have different implementations for these two cases, and the network can't mix flat and regular methods.

7 BEYOND THE CORE SIMULATOR

Writing down all the parameters and connectivity details of a complex network can be a daunting task, even with a simulator handling the basic creation and simulation routines. Draculab's general approach is to separate the core simulator from the tools used to make simulations easier to write and visualize.

Python provides enough power so that individual users can create configuration and visualization solutions according to the model they are writing. Nevertheless, we offer a few tools to make this easier. First and foremost is the `topology` module, which allows to create spatially structured connections. The second are the `ei_net` and `ei_network` modules. These are a collection of Python classes and methods to quickly create a particular type of networks with standard parameters, and to visualize the simulation results using Matplotlib.

Draculab's `topology` model is inspired in NEST's topology module [1]. Although the tools to define connection profiles are similar, there are some key differences, the main one being the Draculab's `topology` module does not use layer objects.

The function used to create structured connections is called `topo_connect`. As with `network.connect`, the first argument to `topo_connect` is a list with the IDs of the units sending the connections, and the second argument is a list with the IDs of the units receiving the connections. Unlike `network.connect`, all units must have a `coordinates` attribute that describes their spatial location. The third argument is a connection specification dictionary, which determines the probability of connection (and optionally strength of synaptic weights) between any two units based on their coordinates. The fourth argument is a synapse specification dictionary, with the same format as in `network.connect`.

Since `topo_connect` expects units with spatial coordinates, the `topology` module also includes a method to create them, called `create_group`. This method resembles `network.create`, but it receives an extra `geometry` parameters dictionary as a way to specify how many units to create, and where to put them. Currently `create_group` can only create flat two-dimensional layers with units in a grid or in random arrangements. Still, both `topo_connect` and `create_group` are written to support a possible expansion to 3D coordinates. An alternative to `create_group` is to provide a list with coordinates to `network.create`.

The `ei_net` and `ei_network` modules have classes that can build networks with 3 populations: one with excitatory units, one with inhibitory units, and one with source units. All parameter dictionaries are written into these classes, so networks can be created with a single instruction. After creation the default parameters can be modified (these modifications get automatically logged), and then the network can be *built*. At this point simulations can be run and results can be visualized, either with plots or with animations. The example network from section 2.2 used this approach at a smaller scale. Specifying simulations by modifying a standard set of parameters greatly reduces their description length, and the possibility of errors.

The difference between `ei_net` and `ei_network` is that `ei_net` specifies a single "layer" object, whereas `ei_network` has tools to create and connect several "layer" elements. Both modules provide solutions to the most common tasks in neural network simulations, including parameter specification, input configuration, storage, documentation, visualization of connections, and visualization of results. These two modules implement a particular type of network for a particular type of simulation, but it is expected that individual users can adapt these tools to their own needs. This complements various other choices that users have, such as neurotools (`http://neuralensemble.org/NeuroTools/`) for simulation management and analysis, and scikit-learn (`https://scikit-learn.org/`) for machine learning analytics.

[1] http://www.nest-simulator.org/introduction-to-pynest/part-4-topologically-structured-networks/

It is easy to program a plant to interact with the network, as long as the plant is simple, such as a planar arm. Things are harder when the plant is more complex, such as the model of a particular robot, and things are downright hard if the actuator has to interact with a virtual environment. Fortunately there are physics simulators written with these considerations in mind Ivaldi et al. (2014). When using an external physics simulator the plant object can stop being an implementation of the physics simulation, and become an interface to the physics simulator. Writing a simulation where Draculab interacts with a physics simulator may require considerable expertise.

A relatively simple option to integrate a Draculab controller in a virtual environment is provided by the HBP neurorobotics platform (NRP) Hinkel et al. (2017) (`https://neurorobotics.net`). The NRP calls a Python *transfer function* on each simulation step, and Draculab can be used in it. In part this is possible because Draculab's core is compatible with Python 2.7, although it was developed with Python 3.5.

The obvious drawback of using the NRP is that the user needs to understand how to program it. Also, a basic understaing of the services provided by the Robot Operating System (`https://www.ros.org/`) may be necessary in some cases. Physics simulators make it easier to produce a virtual environment, but they bring a new learning curve.

8 ALTERNATIVES TO DRACULAB

There are excellent simulators for firing rate networks Rougier and Fix (2012); Aisa et al. (2008); Bekolay et al. (2014); Vitay et al. (2015); Cofer et al. (2010); Tosi and Yoshimi (2016), but it is unusual to find rate simulators that take into account connection delays. To the authors' knowledge there are currently two simulators with this capability: The Virtual Brain Sanz Leon et al. (2013) (`https://www.thevirtualbrain.org`), and NEST Diesmann and Gewaltig (2001) (`http://www.nest-simulator.org/`).

The Virtual Brain is a simulation platform with a graphical user interface. Among other things, this platform includes data management and analysis functionality so that models with realistic large-scale connectivity of identified brain regions can be compared with brain scanning data such as fMRI, MEG, or EEG. Although it is an open source project, it does not appear feasible for normal users to add their own custom models or to integrate the platform with biomechanical simulations.

NEST is a simulator for spiking neural networks, but it has recently incorporated firing rate models Hahne et al. (2017). Over the years NEST has become a mature software project, with a large C++ codebase capable of handling massive parallel simulations on supercomputers. Learning how to create new neuron or synapse models is thus a significant endeavor, but the resulting implementations can take advantage of NEST's infrastructure.

Considering that NEST is often taken as a reference for the performance of spiking neural models, it seems appropriate to use its rate models to assess Draculab's efficiency. NEST's rate models still lack features such as sources of temporally varying inputs or plastic synapses, but we expect that development will continue to move forward. At the time of this writing continuous-time inputs to rate models were soon to be released (personal communication).

As an initial exploration the `rate_neuron_dm` example distributed with the NEST version 2.16 codebase was modified by changing its instantaneous connections to delayed connections with a 4 millisecond delay. The elapsed simulation time was compared with that of an equivalent Draculab

implementation (Appendix D). Both simulations showed the same response in the absence of noise. When the NEST file is modified only by placing a delay in the connections the execution time rises sharply, as seen in table 1. Such a result is restricted to networks with small *resolution* values. This resolution is a kernel parameter in NEST, specifying how much precision is available in the output. The closest parameter in Draculab may be the minimum delay divided by the minimum buffer size, which is matched with NEST's resolution for the comparisons of this section. Reducing the resolution in the `rate_neuron_dm` example dramatically brought down the execution time for NEST (table 1).

Draculab is not designed for large networks, whereas NEST is highly adapted for this. To explore how computation times scale with network size both simulators were programmed with equivalent network simulations (Appendix D). For this comparison, Draculab used a flattened network. Figure 5 shows the results. For small networks both simulators perform similarly, but NEST clearly scales better for large networks.

Draculab's flexibility complements NEST's speed. Researchers who want to work with new models can quickly create and test Draculab prototypes, going through several iterations until the right properties are found. NEST can be used when the need for large simulations arises, or when it is desirable to integrate with spiking networks.

9 DISCUSSION

The inclusion of connection delays in a firing rate simulator merits some consideration.

Research in firing rate neural networks has brought a large number of results. Many of the early results produced the Parallel Distributed Processing Paradigm Rumelhart and Mcclelland (1986) (including backpropagation and Boltzmann machines), eventually leading to the more recent "deep learning" trend Schmidhuber (2015). Because of their feedforward connectivity and lack of continuous dynamics these models have deviated from biological plausibility, and it is unclear whether their computation style is anything like biological brains. There is also a wealth of firing rate models that represent cognitive processes (e.g. O'Reilly (1998); Mastebroek et al. (2001); Eliasmith (2013); Carpenter and Grossberg (1991)). Although these latter models aim for biological plausibility, virtually all of them lack connection delays.

The omission of connection delays is not surprising. Mathematically it moves the models from the realm of linear algebra into functional analysis, with the consequent increase in complexity. Computationally it forces the storage of past activation values for all units, with the consequent increase in computational costs. And then again, delays may not change the network's dynamics, considering that in the brain delays can be as short as one millisecond. Indeed oftentimes delays in the millisecond range don't change the network's dynamics, but it can't be denied that this is not always the case.

In the realm of spiking networks many arguments have been presented to support the importance of precise timing Rieke (1999), in which connection delays play a crucial part (e.g. Izhikevich (2006)). On the other hand, the hypothesis of firing rate coding seems to be the opposite of precise timing playing an important role. This last characterization of firing rate is incorrect. When a network's dynamical system has nonlinearities such as the ones creating a separatrix in the phase plane Ermentrout and Terman (2010), minute changes in the firing rate can create very different dynamical trajectories, and this translates in minute changes in timing causing very different results. This compounds when the firing rate can vary quickly, (e.g. when it comes from a population average, or when its time window for averaging is short).

Precise timing can be important in firing rate networks, it's just that the rate is invariant to permutations in the identity of the neuron producing each individual spike (in population averages).

It is thus clear that in principle the connection delays can play an important role in firing rate networks too, and this role comes into focus when simulations aim to produce biologically plausible closed-loop control in continuous time. Firstly, it is common knowledge in control theory that the presence of delays in closed-loop feedback control imposes fundamental limitations on performance Mirkin and Palmor (2005). These limitations prompted the rise of predictive control. As an example of how this is important, the cerebellum has been hypothesized to be a predictor that permits motor control signals with millisecond precision despite delays in the sensorimotor loop Wolpert et al. (1998); Imamizu and Kawato (2009); Dean and Porrill (2008); Verduzco-Flores and O'Reilly (2015). Secondly, firing rate coding is found in sensory receptors Butts and Goldman (2006); Salinas et al. (2000); Ahissar et al. (2000), and is the way that motor neurons control contractions of skeletal muscle Botelho (1955); Ali et al. (1970); MilnerBrown et al. (1973). Rate coding is thus often found in the interaction of the vertebrate nervous system with its sensors and with its actuators.

This type of considerations are not new to neural modelers. For example, there were plans to provide built-in time-delayed connections in the DANA simulator in order to provide biological realism Rougier and Fix (2012), and the NEST simulator is developing its own infrastructure for this end Hahne et al. (2017). Draculab is also a response to this need, but it also has an unusual aim: that most users can work at the source code level when they need something completely different. The extent to which this can be done depends on the clarity of the design, as well as the user's sophistication level. It is plausible that Python is clear enough, Draculab is simple enough, and researchers in computational neuroscience are capable enough.

Draculab embodies particular research principles. That neural networks are more interesting when they can implement cognitive functions, and when they can do it with biological plausibility. That cognitive models are more illuminating when they span the action-perception loop. Those principles are the source of Draculab's characteristics, such as rate models, delays, and plants.

Future development plans include:

- performance optimizations (without adding too much complexity to the simulator core),
- improvements to the unit tests,
- improvements to the documentation,
- improving the interface with the neurorobotics platform.

Regarding this last point, the HBP neurorobotics platform will provide a new API to integrate it with neural simulators (personal communication). This may be better than the current method, which embeds Draculab in a transfer function. Developers of the neurorobotics platform have also begun translating several OpenSim Delp et al. (2007) (`http://opensim.stanford.edu/`) models into the NRP, making it appropriate for biomechanic simulations.

A final point to make is in regard to the accuracy of the solutions. This is of course dependent on the solver and the parameters used (particularly `rtol` and `atol` for the `odeint` and `solve_ivp` solvers), and on the minimum buffer size (`min_buff_size` network parameter), which controls the accuracy of the interpolation used to obtain past activation values. Draculab's standard configuration uses linear interpolation. Substituting one line in the `unit.get_act` method permits to use Scipy's `interp1d` function instead, but this comes with a reduction in speed.

There is currently no agreed standard to test the accuracy of neural simulators. In the absence of a common benchmark, the approach taken to verify that solutions were correct was to compare runs of the same network in different simulators. In particular, the same network was simulated using XPP Ermentrout (2012) and Simulink (`https://www.mathworks.com/products/simulink.html`), and the resulting unit activities were compared with Draculab. Results were almost identical for the three simulators. This can be found among Draculab's unit tests. Additionally, the performance comparison with NEST (section 8) provided an opportunity to compare Draculab's and NEST's output. Results generally agree, although there are small differences for networks with many units, perhaps from differences in the way that initial conditions are specified.

Draculab's source code can be obtained from this repository: [repository's URL]. It is freely distributed, with the hope that it can become a valuable tool for researchers who find that firing rate models with delays and closed-loop control hit a sweet spot of biological plausibility, technical feasibility, and scientific insight.

CONFLICT OF INTEREST STATEMENT

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

AUTHOR CONTRIBUTIONS

SV created the software, and wrote the first draft of the manuscript. ED supervised the work, and performed manuscript revisions.

SUPPLEMENTAL DATA

Supplementary Material should be uploaded separately on submission, if there are Supplementary Figures, please include the caption in the same file as the figure. LaTeX Supplementary Material templates can be found in the Frontiers LaTeX folder.

DATA AVAILABILITY STATEMENT

The datasets [GENERATED/ANALYZED] for this study can be found in the [NAME OF REPOSITORY] [LINK].

REFERENCES

Ahissar, E., Sosnik, R., and Haidarliu, S. (2000). Transformation from temporal to rate coding in a somatosensory thalamocortical pathway. *Nature* 406, 302–306. doi:10.1038/35018568

Aisa, B., Mingus, B., and O'Reilly, R. (2008). The Emergent neural modeling system. *Neural Networks* 21, 1146–1152. doi:10.1016/j.neunet.2008.06.016

Ali, H. H., Utting, J. E., and Gray, C. (1970). STIMULUS FREQUENCY IN THE DETECTION OF NEUROMUSCULAR BLOCK IN HUMANS. *British Journal of Anaesthesia* 42, 967–978. doi:10.1093/bja/42.11.967

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The Best of Both Worlds. *Computing in Science Engineering* 13, 31–39. doi:10.1109/MCSE.2010.118

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., et al. (2014). Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics* 7. doi:10.3389/fninf.2013.00048

Botelho, S. Y. (1955). Comparison of simultaneously recorded electrical and mechanical activity in myasthenia gravis patients and in partially curarized normal humans. *The American Journal of Medicine* 19, 693–696. doi:10.1016/S0002-9343(55)80010-1

Butts, D. A. and Goldman, M. S. (2006). Tuning Curves, Neuronal Variability, and Sensory Coding. *PLOS Biology* 4, e92. doi:10.1371/journal.pbio.0040092

Carpenter, G. A. and Grossberg, S. (1991). *Pattern Recognition by Self-organizing Neural Networks* (MIT Press). Google-Books-ID: 2u1fH0mxfz0C

Cofer, D., Cymbalyuk, G., Reid, J., Zhu, Y., Heitler, W. J., and Edwards, D. H. (2010). AnimatLab: A 3d graphics environment for neuromechanical simulations. *Journal of Neuroscience Methods* 187, 280–288. doi:10.1016/j.jneumeth.2010.01.005

Dean, P. and Porrill, J. (2008). Adaptive-filter Models of the Cerebellum: Computational Analysis. *The Cerebellum* 7, 567–571. doi:10.1007/s12311-008-0067-3

Delp, S. L., Anderson, F. C., Arnold, A. S., Loan, P., Habib, A., John, C. T., et al. (2007). OpenSim: Open-Source Software to Create and Analyze Dynamic Simulations of Movement. *IEEE Transactions on Biomedical Engineering* 54, 1940–1950. doi:10.1109/TBME.2007.901024

Destexhe, A. and Contreras, D. (2006). Neuronal Computations with Stochastic Network States. *Science* 314, 85–90. doi:10.1126/science.1127241

Diesmann, M. and Gewaltig, M.-O. (2001). NEST: An environment for neural systems simulations. *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis* 58, 43–70

Eliasmith, C. (2013). *How to Build a Brain: A Neural Architecture for Biological Cognition* (OUP USA). Google-Books-ID: BK0YRJPmuzgC

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2009). PyNEST: a convenient interface to the NEST simulator. *Frontiers in Neuroinformatics* 2. doi:10.3389/neuro.11.012.2008

Ermentrout, B. (2012). XPPAUT. In *Computational Systems Neurobiology*, ed. N. Le Novère (Dordrecht: Springer Netherlands). 519–531. doi:10.1007/978-94-007-3858-4_17

Ermentrout, G. B. and Terman, D. H. (2010). *Mathematical Foundations of Neuroscience* (Springer Science & Business Media). Google-Books-ID: 0fLdzaFgtjcC

Goodman, D. F. M. and Brette, R. (2009). The Brian simulator. *Frontiers in Neuroscience* 3. doi:10.3389/neuro.01.026.2009

Hahne, J., Dahmen, D., Schuecker, J., Frommer, A., Bolten, M., Helias, M., et al. (2017). Integration of Continuous-Time Dynamics in a Spiking Neural Network Simulator. *Frontiers in Neuroinformatics* 11. doi:10.3389/fninf.2017.00034

Hinkel, G., Groenda, H., Krach, S., Vannucci, L., Denninger, O., Cauli, N., et al. (2017). A Framework for Coupled Simulations of Robots and Spiking Neuronal Networks. *Journal of Intelligent & Robotic Systems* 85, 71–91. doi:10.1007/s10846-016-0412-6

Imamizu, H. and Kawato, M. (2009). Brain mechanisms for predictive control by switching internal models: implications for higher-order cognitive functions. *Psychological Research PRPF* 73, 527–544. doi:10.1007/s00426-009-0235-1

Ivaldi, S., Peters, J., Padois, V., and Nori, F. (2014). Tools for simulating humanoid robot dynamics: A survey based on user feedback. In *2014 IEEE-RAS International Conference on Humanoid Robots*. 842–849. doi:10.1109/HUMANOIDS.2014.7041462

Izhikevich, E. M. (2006). Polychronization: Computation with Spikes. *Neural Computation* 18, 245–282. doi:10.1162/089976606775093882

Law, C. C. and Cooper, L. N. (1994). Formation of receptive fields in realistic visual environments according to the Bienenstock, Cooper, and Munro (BCM) theory. *Proceedings of the National Academy of Sciences* 91, 7797–7801. doi:10.1073/pnas.91.16.7797

London, M. and Häusser, M. (2005). DENDRITIC COMPUTATION. *Annual Review of Neuroscience* 28, 503–532. doi:10.1146/annurev.neuro.28.061604.135703

Mastebroek, H. A., Vos, J. E., and Vos, J. E. (2001). *Plausible Neural Networks for Biological Modelling* (Springer Science & Business Media). Google-Books-ID: Poc3cLw4WOoC

MilnerBrown, H. S., Stein, R. B., and Yemm, R. (1973). Changes in firing rate of human motor units during linearly changing voluntary contractions. *The Journal of Physiology* 230, 371–390. doi:10.1113/jphysiol.1973.sp010193

Mirkin, L. and Palmor, Z. J. (2005). Control Issues in Systems with Loop Delays. *Handbook of Networked and Embedded Control Systems* , 627–648doi:10.1007/0-8176-4404-0_27

Oja, E. (1982). Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology* 15, 267–273. doi:10.1007/BF00275687

O'Reilly, R. C. (1998). Six principles for biologically based computational models of cortical cognition. *Trends in Cognitive Sciences* 2, 455–462. doi:10.1016/S1364-6613(98)01241-8

Porr, B. and Wörgötter, F. (2006). Strongly Improved Stability and Faster Convergence of Temporal Sequence Learning by Using Input Correlations Only. *Neural Computation* 18, 1380–1412. doi:10.1162/neco.2006.18.6.1380

Rieke, F. (1999). *Spikes: Exploring the Neural Code* (MIT Press). Google-Books-ID: 7Vb9huusbcIC

Rougier, N. P. and Fix, J. (2012). DANA: Distributed numerical and adaptive modelling framework. *Network: Computation in Neural Systems* 23, 237–253. doi:10.3109/0954898X.2012.721573

Rumelhart, D. E. and Mcclelland, J. L. (1986). Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations

Salinas, E., Hernández, A., Zainos, A., and Romo, R. (2000). Periodicity and Firing Rate As Candidate Neural Codes for the Frequency of Vibrotactile Stimuli. *Journal of Neuroscience* 20, 5503–5515. doi:10.1523/JNEUROSCI.20-14-05503.2000

Sanz Leon, P., Knock, S. A., Woodman, M. M., Domide, L., Mersmann, J., McIntosh, A. R., et al. (2013). The Virtual Brain: a simulator of primate brain network dynamics. *Frontiers in Neuroinformatics* 7. doi:10.3389/fninf.2013.00010

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks* 61, 85–117. doi:10.1016/j.neunet.2014.09.003

Swain, P. S. and Longtin, A. (2006). Noise in genetic and neural networks. *Chaos: An Interdisciplinary Journal of Nonlinear Science* 16, 026101. doi:10.1063/1.2213613

Tosi, Z. and Yoshimi, J. (2016). Simbrain 3.0: A flexible, visually-oriented neural network simulator. *Neural Networks* 83, 1–10. doi:10.1016/j.neunet.2016.07.005

Verduzco-Flores, S. O. and O'Reilly, R. C. (2015). How the credit assignment problems in motor control could be solved after the cerebellum predicts increases in error. *Frontiers in Computational Neuroscience* 9. doi:10.3389/fncom.2015.00039

Vitay, J., Dinkelbach, H. , and Hamker, F. H. (2015). ANNarchy: a code generation approach to neural simulations on parallel hardware. *Frontiers in Neuroinformatics* 9. doi:10.3389/fninf.2015.00019

Wolpert, D. M., Miall, R. C., and Kawato, M. (1998). Internal models in the cerebellum. *Trends in Cognitive Sciences* 2, 338–347. doi:10.1016/S1364-6613(98)01221-2

10 APPENDIX A

Full source code for the initial example

```
from draculab import *
import numpy as np
import matplotlib.pyplot as plt

# create a network object
net_params = {'min_delay': 0.1, # minimum connection delay
              'min_buff_size': 10 } # minimum buffer size
net = network(net_params) # creates the network
# create units in the network
sig_params = {'type': unit_types.sigmoidal, # unit model
              'init_val': 0.5, # initial value for all sigmoidal units
              'slope': 1,   # slope of the sigmoidal function
              'thresh': 0., # threshold of the sigmoidal function
              'tau': 0.2,   # time constant of the sigmoidal unit
              'tau_fast': 0.1 } # time constant of the 'fast' low-pass filter
inp_params = {'type': unit_types.source, # source units provide inputs
              'init_val': 1.,
              'function': lambda t: np.cos(t) } # a cosine function
sig_units = net.create(10, sig_params) # create 10 sigmoidal units
inp_unit = net.create(1, inp_params) # create 1 input unit
# create connections between the sigmoidal units
conn_spec = {'rule': 'fixed_outdegree', # rule to create connections
             'outdegree': 2, # each unit sends 2 projections randomly
             'delay': 0.2 } # all connections have a delay of .2 time units
syn_spec = {'type': synapse_types.oja, # synapses use the Oja learning rule
            'init_w': {'distribution': 'uniform', 'low': 0.1, 'high': 1.},
            'lrate': 0.1 } # learning rate for the Oja rule
net.connect(sig_units, sig_units, conn_spec, syn_spec) # create the connections
# connect the input to the sigmoidal units
conn_spec_2 = {'rule': 'all_to_all', # rule to create connections
               'delay': 0.1 } # all connections have a delay of .1 time units
syn_spec_2 = {'type': synapse_types.static, # synapses don't change
              'init_w': 0.5 } # all synapses have this initial weight
net.connect(inp_unit, sig_units, conn_spec_2, syn_spec_2) # create the connections
# run the simulation for 10 time units
sim_data = net.run(10.)
# visualize using Matplotlib
sig_activs = np.array(sim_data[1])[sig_units]
plt.plot(sim_data[0], sig_activs.transpose())
plt.show()
```

11 APPENDIX B

The `linear` unit class.

```
class linear(unit):
   """ An implementation of a linear unit.

      The output approaches the sum of the inputs multiplied by their
      synaptic weights, evolving with a time constant 'tau'.
   """
   def __init__(self, ID, params, network):
      """ The unit constructor.

      Args:
         ID, params, network: same as in the parent's constructor.
         In addition, params should have the following entries.
            REQUIRED PARAMETERS
            'tau' : Time constant of the update dynamics.
      """
      unit.__init__(self, ID, params, network)
      self.tau = params['tau'] # the time constant of the dynamics

   def derivatives(self, y, t):
      """ This function returns the derivatives of the state variables
         at a given point in time.

         Args:
            y: a 1-element array or list with the current firing rate.
            t: time when the derivative is evaluated.
      """
      return( self.get_input_sum(t) - y[0] ) / self.tau
```

12 APPENDIX C

The `network.run` method.

```
def run(self, total_time):
   """
   Simulate the network for the given time.

   This method takes steps of 'min_delay' length, in which the units, synapses
   and plants use their own methods to advance their state variables.

   After run(T) is finished, calling run(T) again continues the simulation
   starting at the last state of the previous simulation.
```

```
875
        Args:
           total_time: time that the simulation will last.

        Returns:
           The method returns a 3-tuple (times, unit_store, plant_store):
           times: a numpy array with the simulation times when the update functions
                 were called. These times will begin at the initial simulation time,
                 and advance in 'min_delay' increments until 'total_time' is completed.
           unit_store: a list of numpy arrays. unit_store[i][j] contains the activity
                    of the i-th unit at time j-th timepoint (e.g. at times[j]).
           plant_store: a list of 2-dimensional numpy arrays. plant_store[i][j][k] is
                     the value of the k-th state variable, at the j-th timepoint,
                     for the i-th plant.
        """
890     Nsteps = int(total_time/self.min_delay) # total number of simulation steps
        # arrays to store unit activities. np stands for numpy
        unit_store = [np.zeros(Nsteps) for i in range(self.n_units)]
        # arrays to store plant steps
        plant_store = [np.zeros((Nsteps,p.dim)) for p in self.plants]
895     # array to store initial times of simulation steps
        times = np.zeros(Nsteps) + self.sim_time

        for step in range(Nsteps):
           times[step] = self.sim_time # self.sim_time persists between calls
900                                    # to network.run()
           # store current unit activities
           for uid, unit in enumerate(self.units):
              unit_store[uid][step] = unit.get_act(self.sim_time)

905        # store current plant state variables
           for pid, plant in enumerate(self.plants):
              plant_store[pid][step,:] = plant.get_state(self.sim_time)

           # update units
910        for unit in self.units:
              unit.update(self.sim_time)

           # update plants
           for plant in self.plants:
915           plant.update(self.sim_timef

           self.sim_time += self.min_delay

        return times, unit_store, plant_store
```

13 APPENDIX D

Comparing with NEST

The file `rate_neuron_dm.py` distributed with the source code of NEST 2.16 was modified to use connection delays, and to store the data for the simulations with no noise. The resulting code is shown below (comments removed).

```
import nest
import pylab
import numpy

def build_network(sigma, dt):
   nest.ResetKernel()
   nest.SetKernelStatus({'resolution': dt, 'use_wfr': False})
   Params = {'lambda': 0.1,
           'std': sigma,
           'tau': 1.,
           'rectify_output': True }
   D1 = nest.Create('lin_rate_ipn', params=Params)
   D2 = nest.Create('lin_rate_ipn', params=Params)
   nest.Connect(D1, D2, 'all_to_all', {
      'model': 'rate_connection_delayed', 'weight': -0.2, 'delay': 4.})
   nest.Connect(D2, D1, 'all_to_all', {
      'model': 'rate_connection_delayed', 'weight': -0.2, 'delay': 4.})
   mm = nest.Create('multimeter')
   nest.SetStatus(mm, {'record_from': ['rate']})
   nest.Connect(mm, D1, syn_spec={'delay': dt})
   nest.Connect(mm, D2, syn_spec={'delay': dt})

   return D1, D2, mm


fig_size = [14, 8]
fig_rows = 3
fig_cols = 3
fig_plots = fig_rows * fig_cols
face = 'white'
edge = 'white'
ax = [None] * fig_plots
fig = pylab.figure(facecolor=face, edgecolor=edge, figsize=fig_size)

dt = 1e-3
print('dt = ' + str(dt))
```

```
sigma = [0.0, 0.1, 0.2]
dE = [0.0, 0.004, 0.008]
T = numpy.linspace(0, 200, 199)
noiseless_data = [[],[],[]]
for i in range(9):
   c = i % 3
   r = int(i / 3)
   D1, D2, mm = build_network(sigma[r], dt)
   nest.Simulate(100.0)
   nest.SetStatus(D1, {'mean': 1. + dE[c]})
   nest.SetStatus(D2, {'mean': 1. - dE[c]})
   nest.Simulate(100.0)

   data = nest.GetStatus(mm)
   senders = data[0]['events']['senders']
   voltages = data[0]['events']['rate']
   if sigma[r] == 0.:
      noiseless_data[c] = data

   ax[i] = fig.add_subplot(fig_rows, fig_cols, i + 1)
   ax[i].plot(T, voltages[numpy.where(senders == D1)],
            'b', linewidth=2, label="D1")
   ax[i].plot(T, voltages[numpy.where(senders == D2)],
            'r', linewidth=2, label="D2")
   ax[i].set_ylim([-.5, 12.])
   ax[i].get_xaxis().set_ticks([])
   ax[i].get_yaxis().set_ticks([])
   if c == 0:
      ax[i].set_ylabel("activity␣($\sigma=%.1f$)␣" % (sigma[r]))
      ax[i].get_yaxis().set_ticks([0, 3, 6, 9, 12])

   if r == 0:
      ax[i].set_title("$\Delta␣E=%.3f$␣" % (dE[c]))
      if c == 2:
         pylab.legend(loc=0)
   if r == 2:
      ax[i].get_xaxis().set_ticks([0, 50, 100, 150, 200])
      ax[i].set_xlabel('time␣(ms)')

pylab.show()
```

A Draculab unit model that replicates the dynamics of the `lin_rate_ipn` model was created, and specified to use the stochastic exponential Euler numerical solver when its parameter `lambda` was

positive. The code from `rate_neuron_dm.py` was modified to run simulations with this model instead. The resulting code is below.

```
from draculab import *
import pylab
import numpy as np

def build_network(sigma, dt):
    net_params = {'min_delay': 1., # 1 ms
                  'min_buff_size': int(np.ceil(1./dt)) }
    net = network(net_params)
    unit_params = {'type': unit_types.noisy_linear,
                   'init_val': 0.,
                   'tau': 1.,   # time constant in ms
                   'lambda': 0.1, # passive decay rate
                   'mu': 0.,    # mean of Gaussian noise
                   'sigma': sigma} # Std. Dev. of Gaussian noise
    D1 = net.create(1, unit_params)
    D2 = net.create(1, unit_params)
    conn_spec = {'rule': 'all_to_all',
                 'delay': 4.}
    syn_spec = {'type': synapse_types.static,
                'init_w': -.2}
    net.connect(D1, D2, conn_spec, syn_spec)
    net.connect(D2, D1, conn_spec, syn_spec)

    return net, D1[0], D2[0]

fig_size = [14, 8]
fig_rows = 3
fig_cols = 3
fig_plots = fig_rows * fig_cols
face = 'white'
edge = 'white'
ax = [None] * fig_plots
fig = pylab.figure(facecolor=face, edgecolor=edge, figsize=fig_size)

dt = 1e-3
sigma = [0.0, 0.1, 0.2]
dE = [0.0, 0.004, 0.008]
noiseless_activs = [[], [], []]
for i in range(9):
    c = i % 3
    r = int(i / 3)
    net, D1, D2 = build_network(sigma[r], dt)
```

```
    times, activs, _ = net.run(100.)
    net.units[D1].mu = 1. + dE[c]
    net.units[D2].mu = 1. - dE[c]
    times2, activs2, _ = net.run(100.)

    times = np.append(times, times2)
    activs = np.concatenate((activs, activs2), axis=1)
    if sigma[r] == 0.:
       noiseless_activs[c] = activs
    ax[i] = fig.add_subplot(fig_rows, fig_cols, i + 1)
    ax[i].plot(times, activs[D1], 'b', linewidth=2, label="D1")
    ax[i].plot(times, activs[D2], 'r', linewidth=2, label="D2")
    ax[i].set_ylim([-.5, 12.])
    ax[i].get_xaxis().set_ticks([])
    ax[i].get_yaxis().set_ticks([])
    if c == 0:
       ax[i].set_ylabel("activity␣($\sigma=%.1f$)␣" % (sigma[r]))
       ax[i].get_yaxis().set_ticks([0, 3, 6, 9, 12])

    if r == 0:
       ax[i].set_title("$\Delta␣E=%.3f$␣" % (dE[c]))
       if c == 2:
          pylab.legend(loc=0)
    if r == 2:
       ax[i].get_xaxis().set_ticks([0, 50, 100, 150, 200])
       ax[i].set_xlabel('time␣(ms)')
pylab.show()
```

The comparison leading to figure 5 was produced with a Jupyter notebook whose most relevant cells are shown next. The full code can be obtained from the first author (SV) upon request. The simulation was run in a personal workstation with an Intel Core i9-7900X processor.

```
import nest
import nest.topology as topp
import draculab
import numpy as np
import pylab
from matplotlib import pyplot as plt
import time
import cProfile
import pstats

# --- NEST network ---
class nest_runner():
```

```
    def __init__(self, h=.1, N=10,threads=1):
        #h = .1 # ms resolution
        #N = 10 # number of rows and columns
        nest.ResetKernel()
        nest.SetKernelStatus({'resolution': h, 'use_wfr':False,
                              'local_num_threads' : threads})

        # create rate units and recording device
        lin_params = {'linear_summation': True,
              'mean': 0., # mean of the Gaussian white noise
              'std': 0., # standard deviation of the Gaussian white noise
              'tau': 20., # unit's time constant (milliseconds)
              'g': 1.,  # unit's gain
              'lambda': 1., # decay rate
              'rectify_output': True,
              'mult_coupling' : False,
              'rate': 0.} # initial rate
        nest.CopyModel('lin_rate_ipn', 'linear_rate', params=lin_params)

        layer_dict = {"extent" : [float(N), float(N)], # the size of the layer in mm
             "rows" : N, # the number of rows in this layer ...
             "columns" : N, # ... and the number of columns
             "edge_wrap" : True, # periodic boundaries
             "elements" : "linear_rate"} # the element at each (x,y)
                                         # coordinate in the grid
        self.lin_layer = topp.CreateLayer(layer_dict)
        conn_spec = {"connection_type":"divergent",
            "mask": {"circular":{"radius":1.5}},
            "kernel": 1.,
            "synapse_model": 'rate_connection_delayed',
            "delays": {'linear': {'c':2., 'a':.5}},
            "weights": .1,
            "allow_autapses": False }
        topp.ConnectLayers(self.lin_layer, self.lin_layer, conn_spec)
        # create/connect multimeter
        self.mm = nest.Create('multimeter')
        nest.SetStatus(self.mm, {'record_from': ['rate']})
        self.recorded = nest.GetNodes(self.lin_layer)[0]
        #recorded = topp.GetElement(self.lin_layer, [0,0])
        nest.Connect(self.mm, self.recorded, syn_spec={'delay':h})
    # create a sinusoidal input function
    def inp(self, amp, freq, t):
        return amp * np.sin(2.*freq*np.pi*(t))
    def run(self, sim_time):
```

```
        # simulate in a loop
        step_len = .1 # duration of constant input
        n_steps = int(sim_time/step_len)
        inp_amp = -1.0
        inp_freq = 0.02
        stimd = self.recorded[0::2]
        start = time.time()
        for step in range(n_steps):
            nest.SetStatus(stimd, {'mean': self.inp(inp_amp, inp_freq, step*step_len)})
            nest.Simulate(step_len)
        end = time.time()
        elapsed = end-start
        print('Execution time: %f' % (elapsed))
        return elapsed, nest.GetStatus(self.mm)

    def plot(self, data):
        #data = nest.GetStatus(self.mm)
        senders = data[0]['events']['senders']
        rates = data[0]['events']['rate']
        times = data[0]['events']['times']
        T = times[np.where(senders == self.recorded[0])]
        fig = pylab.figure()
        pylab.plot(T, rates[np.where(senders == self.recorded[0])])
        pylab.plot(T, rates[np.where(senders == self.recorded[1])])
        pylab.plot(T, rates[np.where(senders == self.recorded[2])])
        pylab.show()


# --- draculab network ---
class dracu_runner():

    def __init__(self, h, N):
        net_params = {'min_delay': 1., # 1 ms
                      'min_buff_size': int(1./h) } # h=nest's resolution
        self.net = draculab.network(net_params)
        # create rate units and recording device
        # N = number of rows and columns defined above
        lin_params = {'type': draculab.unit_types.noisy_linear,
             'init_val': 0., # initial rate
             'tau': 20., # unit's time constant (milliseconds)
             'lambda': 1., # decay rate
             'mu': 0.,   # mean of the Gaussian white noise
             'sigma': 0. }
        source_params = {'type': draculab.unit_types.source,
               'init_val': 0., #0.06, # roughly the nest input in the first ms
```

```
            'coordinates': np.array((0.,0.)),
            'function': lambda t: None }
        geom = { 'shape':'sheet',
            'extent':[N, N], # in millimeters
            'center':[0.,0.],
            'arrangement':'grid',
            'rows':N,
            'columns':N }
        topo = draculab.topology()
        self.lins = topo.create_group(self.net, geom, lin_params)
        conn_dict = {'connection_type': 'divergent',
            'mask': {"circular":{"radius":1.5}},
            'kernel': 1.,
            'delays': {'linear': {'c': 2., 'a':.5}},
            'edge_wrap': True,
            'boundary': {'center': np.array([0.,0.]), 'extent': [N, N]},
            'allow_autapses': False }
        syn_dict = {'type': draculab.synapse_types.static,
                'init_w': 0.1 }
        topo.topo_connect(self.net, self.lins, self.lins, conn_dict, syn_dict)
        self.src = self.net.create(1, source_params)
        self.net.connect(self.src, self.lins[0::2], {'rule':'all_to_all', 'delay':1.},
                    {'type': draculab.synapse_types.static, 'init_w':1.})
    # create a sinusoidal input function
    def inp(self, amp, freq, t):
        return amp * np.sin(2.*freq*np.pi*t)
    def make_inp(self, amp, freq):
        return lambda t: self.inp(amp, freq, t)
    def run(self, sim_time):
        # simulate in a loop
        if __name__=='__main__':
            #sim_time = 100. # simulation time in ms
            step_len = 2. # duration of constant input
            n_steps = int(sim_time/step_len)
            inp_amp = -1.0
            inp_freq = 0.02
            #times = np.array([])
            #activs = np.array([[] for _ in lins+src])
            self.net.units[self.src[0]].set_function(self.make_inp(inp_amp, inp_freq))
            start = time.time()
            times, activs, _ = self.net.run(sim_time)
            end = time.time()
            elapsed = end - start
            print('Execution time: %f' % (elapsed))
            return elapsed, times, activs
```

```
    def plot(self, times, activs):
        figz = plt.figure()
        plt.plot(times, activs[self.lins[0]])
        plt.plot(times, activs[self.lins[1]])
        plt.plot(times, activs[self.lins[2]])
        plt.show()

# compare execution times for various network sizes
h = 0.1
Ns = list(range(3,34,2))
threads = 20
sim_time = 100.
dracu_times = [0]*len(Ns)
nest_times = [0]*len(Ns)
nest_mth_times = [0]*len(Ns)
for i, N in enumerate(Ns):
    nesty = nest_runner(h, N)
    nest_times[i], _ = nesty.run(sim_time)
    #nesty_mth = nest_runner(h, N, threads)
    #nest_mth_times[i], _ = nesty_mth.run(sim_time)
    dracu = dracu_runner(h,N)
    dracu_times[i], _, _ = dracu.run(sim_time)

comp_fig = plt.figure(figsize=(16,8))
ax = comp_fig.gca()
ax.plot([n**2. for n in Ns], dracu_times, label='Draculab', linewidth=4.)
ax.plot([n**2. for n in Ns], nest_times, label='NEST', linewidth=4.)
#ax.plot([n**2. for n in Ns], nest_mth_times,
#        label='NEST '+str(threads)+' threads', linewidth=4.)
ax.set_ylabel('time [s]', fontsize='xx-large')
ax.set_xlabel('number of units', fontsize='xx-large')
ax.tick_params(labelsize=25)
plt.legend(bbox_to_anchor=(0.2,0.9), fontsize='xx-large')
plt.show()
```

Results for NEST with 20 threads were not included in figure 5 because they were almost identical to those with a single thread.

FIGURE CAPTIONS

resolution	0.001	0.01
Draculab time	5.8 s	1.0 s
NEST time	92.5 s	1.5 s

Table 1. Simualtion times for the rate_neuron_dm model.

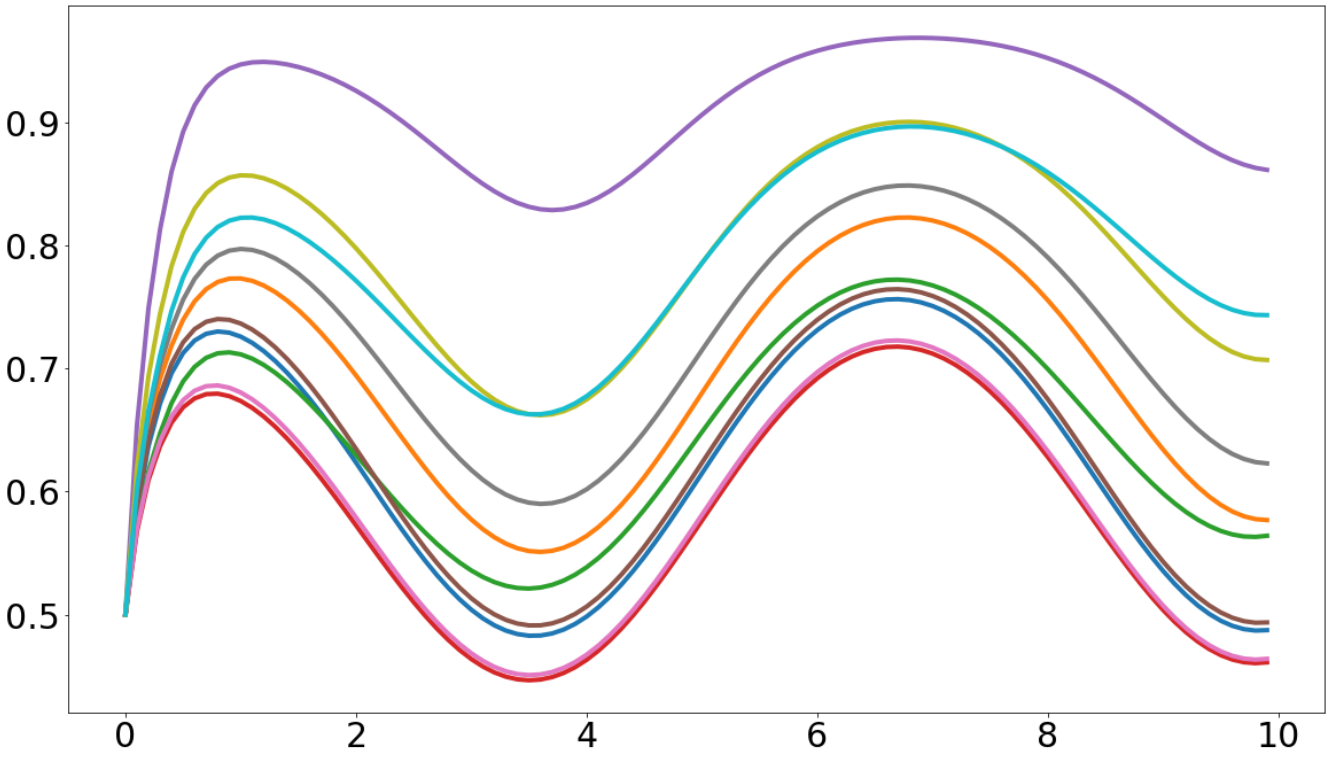

Figure 1. Output of the first example program. It can be seen that the heterogeneous initial values of the synaptic weights cause the units to have different responses.

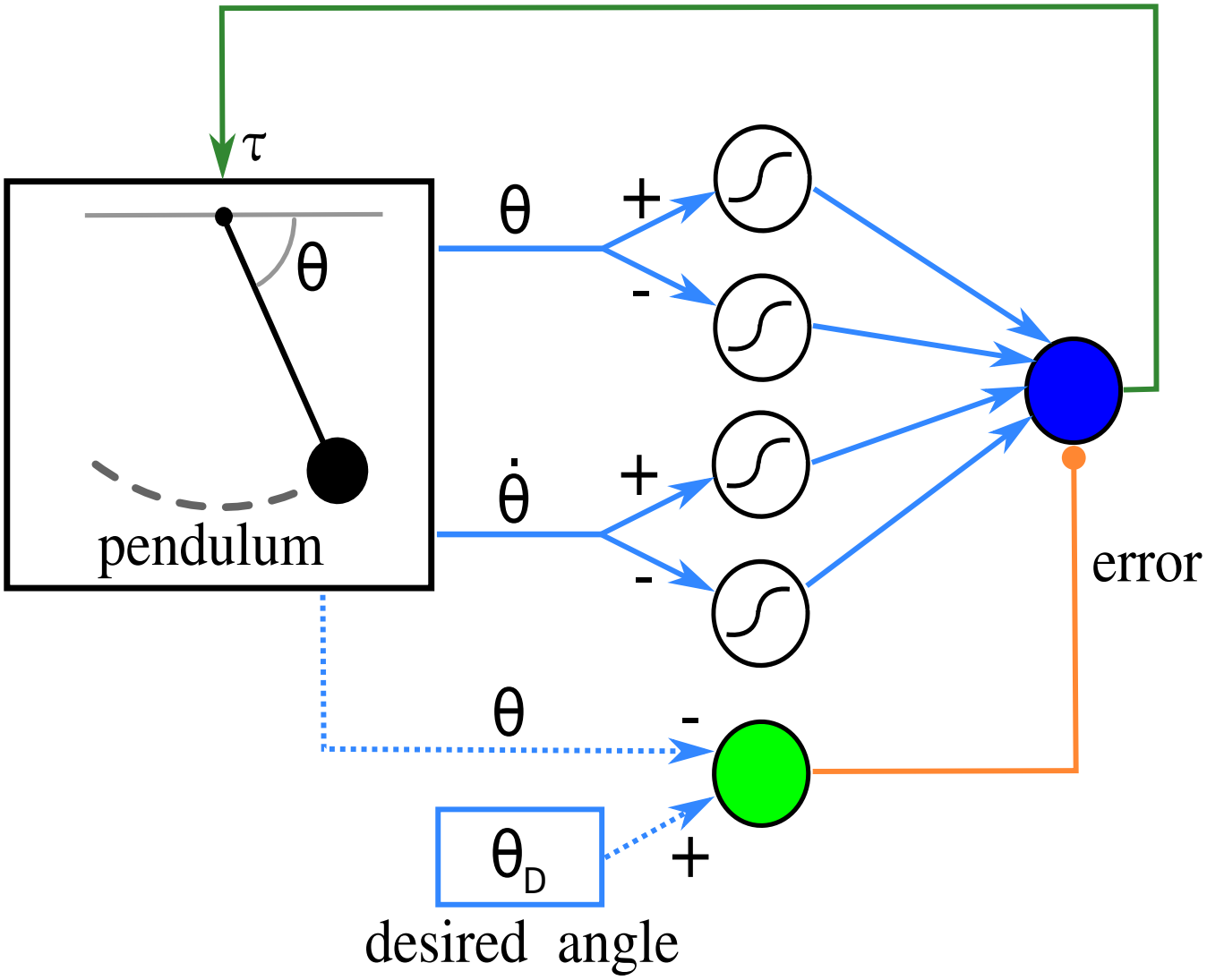

Figure 2. Diagram of the network in the closed-loop simulation. The pendulum sends angle and angular velocity signals to four sigmoidal afferent units, which in turn send inputs to a control unit (blue circle). The control unit produces a torque on the pendulum based on this input. The source unit, in green, calculates an error signal from the desired and the current angle. The derivative of this error is used by the learning rule. Blue dotted lines indicate that the source unit does not receive the angles from connections; instead, its function reads them directly.

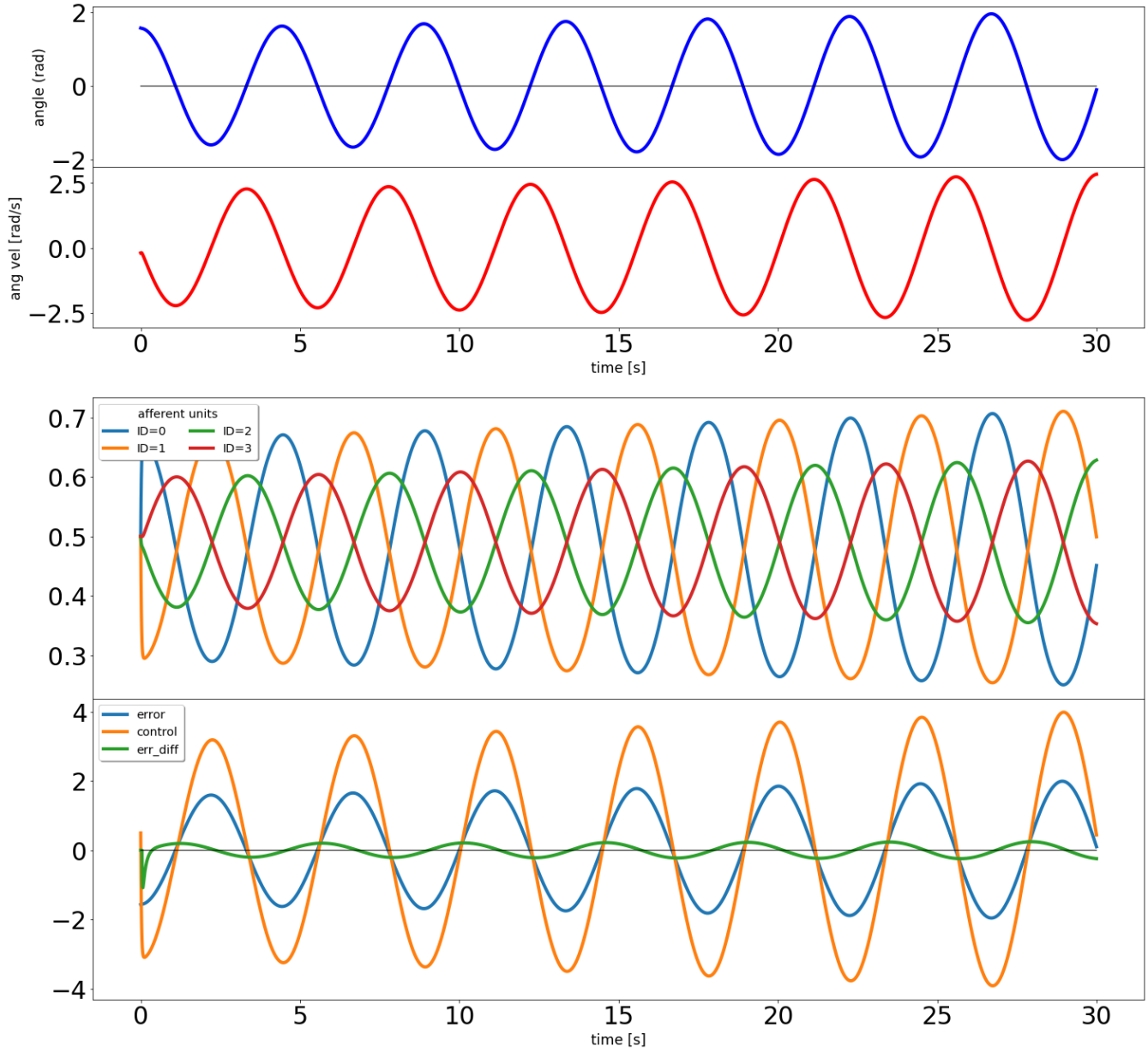

Figure 3. Output of the closed loop simulation using static synaptic weights. The top two plots show the angle and angular velocity of the pendulum. The next plot shows the activity of the four afferent units, and the bottom plot shows the error signal, the activity of the control unit, and a signal proportional to the error's derivative (used in the learning rule).

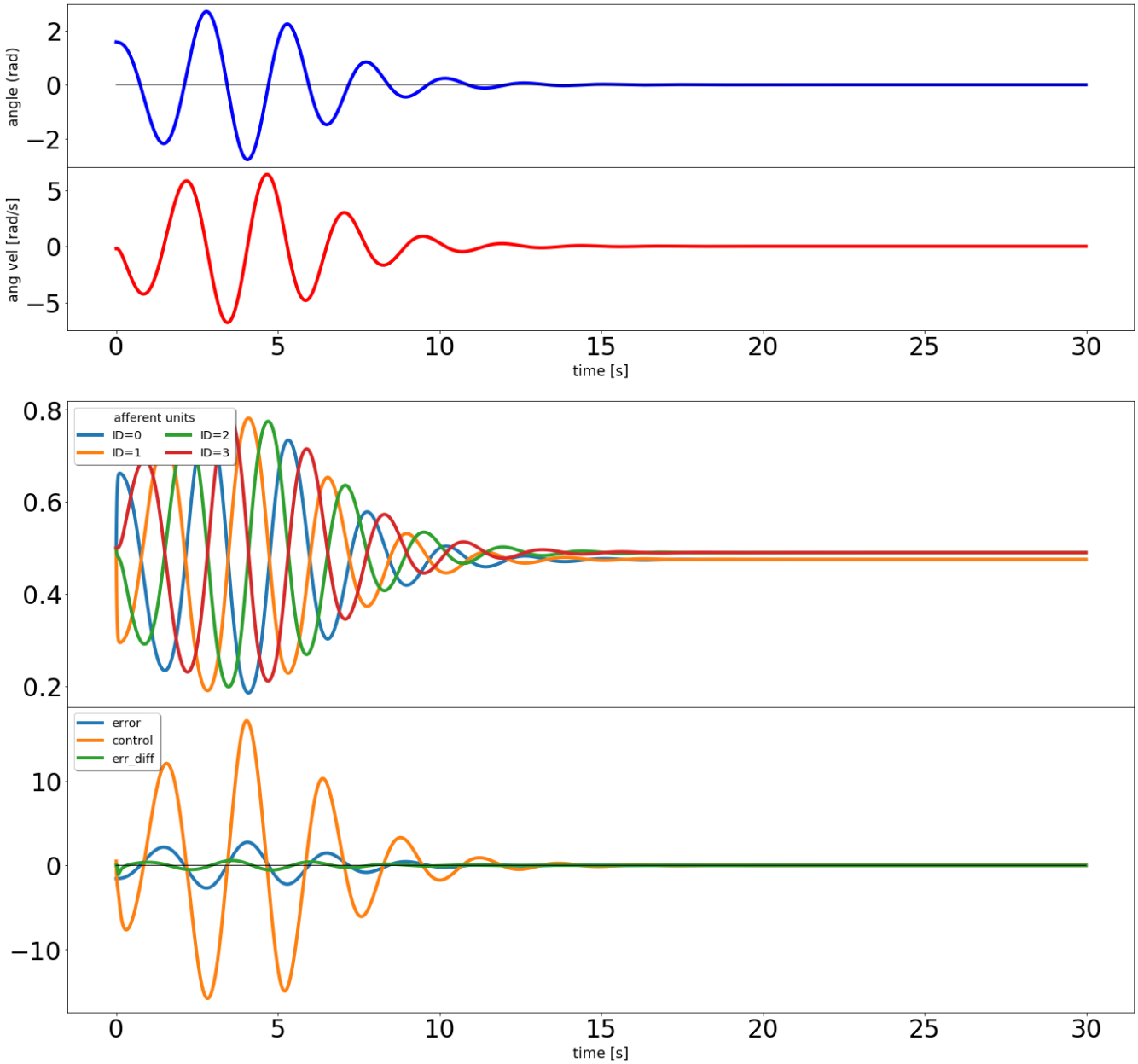

Figure 4. Output of the closed loop simulation using the input correlation rule to adjust the synaptic weights of the afferent inputs to the control unit.

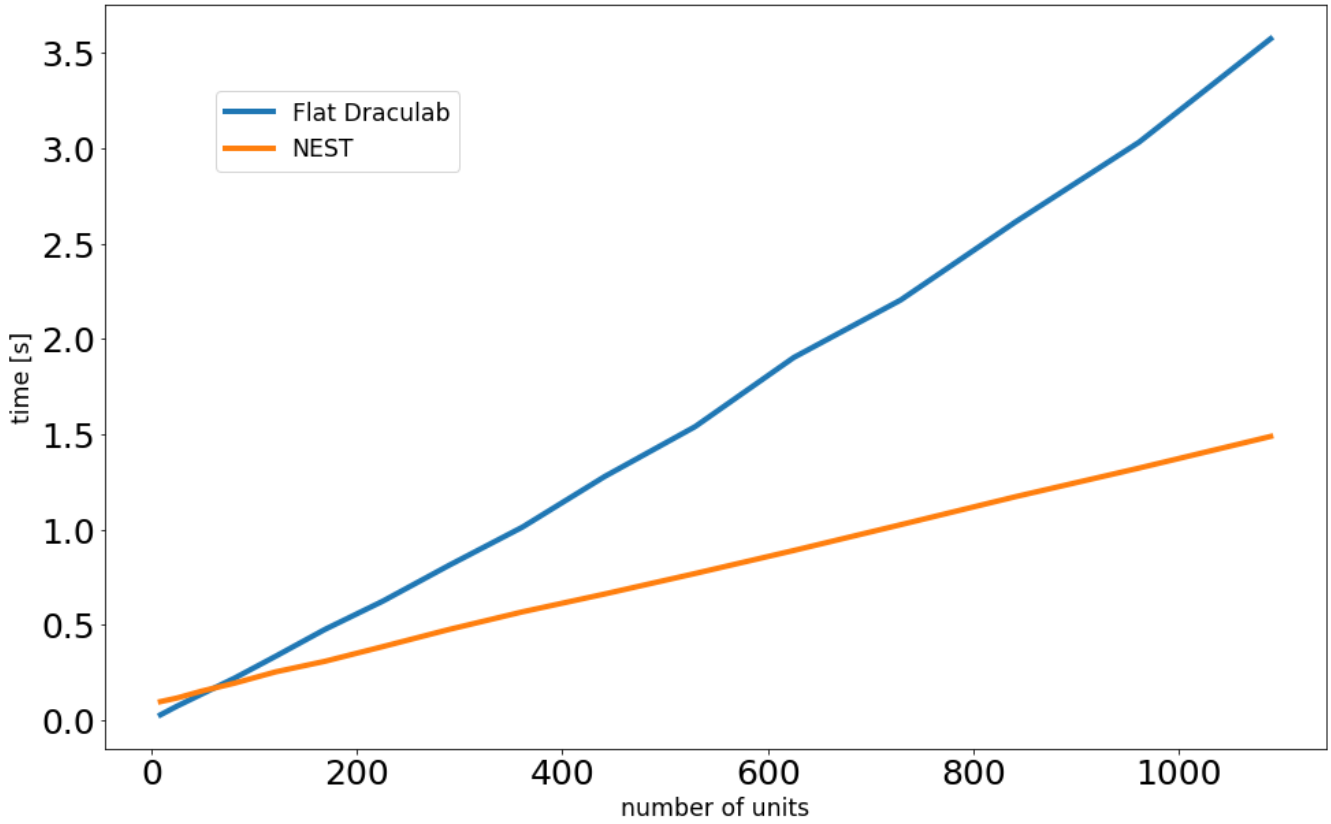

Figure 5. Simulation times as a function of the number of units.