part 2: populations of neurons

introduction

In this handout we look at creating and parameterising batches of `neurons`, and connecting them. When you have worked through this material, you will know how to:

- create populations of neurons with specific parameters
- set model parameters before creation
- define models with customised parameters
- randomise parameters after creation
- make random connections between populations
- set up devices to start, stop and save data to file
- reset simulations

For more information on the usage of PyNEST, please see the other sections of this primer:

- Part 1: Neurons and simple neural networks
- Part 3: Connecting networks with synapses
- Part 4: Topologically structured networks

More advanced examples can be found at Example Networks, or have a look at at the source directory of your NEST installation in the subdirectory: `pynest/examples/`.

creating parameterised populations

of nodes

In the previous handout, we introduced the function `Create(model, n=1, params=None)`. Its mandatory argument is the model name, which determines what type the nodes to be created should be. Its two optional arguments are `n`, which gives the number of nodes to be created (default: 1) and `params`, which is a dictionary giving the parameters with which the nodes should be initialised. So the most basic way of creating a batch of identically parameterised neurons is to exploit the optional arguments of `Create()`:

```
ndict = {"I_e": 200.0, "tau_m": 20.0}
neuronpop = nest.Create("iaf_psc_alpha", 100, params=ndict)
```

The variable `neuronpop` is a list of all the ids of the created neurons.

Parameterising the neurons at creation is more efficient than using `SetStatus()` after creation, so try to do this wherever possible.

We can also set the parameters of a neuron model *before* creation, which allows us to define a simulation more concisely in many cases. If many individual batches of neurons are to be produced, it is more convenient to set the defaults of the model, so that all neurons created from that model will automatically have the same parameters. The defaults of a model can be queried with `GetDefaults(model)`, and set with `SetDefaults(model, params)`, where `params` is a dictionary containing the desired parameter/value pairings. For example:

```
ndict = {"I_e": 200.0, "tau_m": 20.0}
nest.SetDefaults("iaf_psc_alpha", ndict)
neuronpop1 = nest.Create("iaf_psc_alpha", 100)
neuronpop2 = nest.Create("iaf_psc_alpha", 100)
neuronpop3 = nest.Create("iaf_psc_alpha", 100)
```

The three populations are now identically parameterised with the usual model default values for all parameters except `I_e` and `tau_m`, which have the values specified in the dictionary `ndict`.

If batches of neurons should be of the same model but using different parameters, it is handy to use `CopyModel(existing, new, params=None)` to make a customised version of a neuron model with its own default parameters. This function is an effective tool to help you write clearer simulation scripts, as you can use the name of the model to indicate what role it plays in the simulation. Set up your customised model in two steps using `SetDefaults()`:

```
edict = {"I_e": 200.0, "tau_m": 20.0}
nest.CopyModel("iaf_psc_alpha", "exc_iaf_neuron")
nest.SetDefaults("exc_iaf_neuron", edict)
```

or in one step:

```
idict = {"I_e": 300.0}
nest.CopyModel("iaf_psc_alpha", "inh_iaf_neuron", params=idict)
```

Either way, the newly defined models can now be used to generate neuron populations and will also be returned by the function `Models()`.

```
epop1 = nest.Create("exc_iaf_neuron", 100)
epop2 = nest.Create("exc_iaf_neuron", 100)
ipop1 = nest.Create("inh_iaf_neuron", 30)
ipop2 = nest.Create("inh_iaf_neuron", 30)
```

It is also possible to create populations with an inhomogeneous set of parameters. You would typically create the complete set of parameters, depending on experimental constraints, and then create all the neurons in one go. To do this supply a list of dictionaries of the same length as the number of neurons (or synapses) created:

```
parameter_list = [{"I_e": 200.0, "tau_m": 20.0}, {"I_e": 150.0, "tau_m": 30.0}]
epop3 = nest.Create("exc_iaf_neuron", 2, parameter_list)
```

setting parameters for populations of neurons

It is not always possible to set all parameters for a neuron model at or before creation. A classic example of this is when some parameter should be drawn from a random distribution. Of course, it is always possible to make a loop over the population and set the status of each one:

```
Vth=-55.
Vrest=-70.
for neuron in epop1:
    nest.SetStatus([neuron], {"V_m": Vrest+(Vth-Vrest)*numpy.random.rand()})
```

However, `SetStatus()` expects a list of nodes and can set the parameters for each of them, which is more efficient, and thus to be preferred. One way to do it is to give a list of dictionaries which is the same length as the number of nodes to be parameterised, for example using a list comprehension:

```
dVms =  [{"V_m": Vrest+(Vth-Vrest)*numpy.random.rand()} for x in epop1]
nest.SetStatus(epop1, dVms)
```

If we only need to randomise one parameter then there is a more concise way by passing in the name of the parameter and a list of its desired values. Once again, the list must be the same size as the number of nodes to be parameterised:

```
Vms = Vrest+(Vth-Vrest)*numpy.random.rand(len(epop1))
nest.SetStatus(epop1, "V_m", Vms)
```

Note that we are being rather lax with random numbers here. Really we have to take more care with them, especially if we are using multiple threads or distributing over multiple machines. We will worry about this later.

generating populations of neurons with deterministic connections

In the previous handout two neurons were connected using synapse specifications. In this section we extend this example to two populations of ten neurons each.

```python
import pylab
import nest
pop1 = nest.Create("iaf_neuron", 10)
nest.SetStatus(pop1, {"I_e": 376.0})
pop2 = nest.Create("iaf_neuron", 10)
multimeter = nest.Create("multimeter", 10)
nest.SetStatus(multimeter, {"withtime":True, "record_from":["V_m"]})
```

If no connectivity pattern is specified, the populations are connected via the default rule, namely `all_to_all`. Each neuron of `pop1` is connected to every neuron in `pop2`, resulting in 10^2 connections.

```python
nest.Connect(pop1, pop2, syn_spec={"weight":20.0})
```

Alternatively, the neurons can be connected with the `one_to_one`. This means that the first neuron in `pop1` is connected to the first neuron in `pop2`, the second to the second, etc., creating ten connections in total.

```python
nest.Connect(pop1, pop2, "one_to_one", syn_spec={"weight":20.0, "delay":1.0})
```

Finally, the multimeters are connected using the default rule

```python
nest.Connect(multimeter, pop2)
```

Here we have just used very simple connection schemes. Connectivity patterns requiring the specification of further parameters, such as in-degree or connection probabilities, must be defined in a dictionary containing the key `rule` and the key for parameters associated to the rule. Please see Connection management for an illustrated guide to the usage of `Connect`.

connecting populations with random connections

In the previous handout we looked at the connectivity patterns `one_to_one` and `all_to_all`. However, we often want to look at networks with a sparser connectivity than all-to-all. Here we introduce four connectivity patterns which generate random connections between two populations of neurons.

The connection rule `fixed_indegree` allows us to create `n` random connections for each neuron in the target population `post` to a randomly selected neuron from the source population `pre`. The variables `weight` and `delay` can be left unspecified, in which case the default weight and delay are used. Alternatively we can set them in the `syn_spec` , so each created connection has the same weight and delay. Here is an example:

```
d = 1.0
Je = 2.0
Ke = 20
Ji = -4.0
Ki = 12
conn_dict_ex = {"rule": "fixed_indegree", "indegree": Ke}
conn_dict_in = {"rule": "fixed_indegree", "indegree": Ki}
syn_dict_ex = {"delay": d, "weight": Je}
syn_dict_in = {"delay": d, "weight": Ji}
nest.Connect(epop1, ipop1, conn_dict_ex, syn_dict_ex)
nest.Connect(ipop1, epop1, conn_dict_in, syn_dict_in)
```

Now each neuron in the target population `ipop1` has `Ke` incoming random connections chosen from the source population `epop1` with weight `Je` and delay `d`, and each neuron in the target population `epop1` has `Ki` incoming random connections chosen from the source population `ipop1` with weight `Ji` and delay `d`.

The connectivity rule `fixed_outdegree` works in analogous fashion, with `n` connections (keyword `outdegree`) being randomly selected from the target population `post` for each neuron in the source population `pre`. For reasons of efficiency, particularly when simulating in a distributed fashion, it is better to use `fixed_indegree` if possible.

Another connectivity pattern available is `fixed_total_number`. Here `n` connections (keyword `N`) are created by randomly drawing source neurons from the populations `pre` and target neurons from the population `post`.

When choosing the connectivity rule `pairwise_bernoulli` connections are generated by iterating through all possible source-target pairs and creating each connection with the probability `p` (keyword `p`).

In addition to the rule specific parameters `indegree`, `outdegree`, `N` and `p`, the `conn_spec` can contain the keywords `autapses` and `multapses` (set to `False` or `True`) allowing or forbidding self-connections and multiple connections between two neurons, respectively.

Note that for all connectivity rules, it is perfectly legitimate to have the same population simultaneously in the role of `pre` and `post`.

For more information on connecting neurons, please read the documentation of the `Connect` function and consult the guide at Connection management.

specifying the behaviour of devices

All devices implement a basic timing capacity; the parameter `start` (default 0) determines the beginning of the device's activity and the parameter `stop` (default: ∞) its end. These values are taken relative to the value of `origin` (default: 0). For example, the following example creates a `poisson_generator` which is only active between 100 and 150ms:

```
pg = nest.Create("poisson_generator")
nest.SetStatus(pg, {"start": 100.0, "stop": 150.0})
```

This functionality is useful for setting up experimental protocols with stimuli that start and stop at particular times.

So far we have accessed the data recorded by devices directly, by extracting the value of `events`. However, for larger or longer simulations, we may prefer to write the data to file for later analysis instead. All recording devices allow the specification of where data is stored over the parameters `to_memory` (default: `True`), `to_file` (default: `False`) and `to_screen` (default: `False`). The following code sets up a `multimeter` to record data to a named file:

```
recdict = {"to_memory" : False, "to_file" : True, "label" : "epop_mp"}
mm1 = nest.Create("multimeter", params=recdict)
```

If no name for the file is specified using the `label` parameter, NEST will generate its own using the name of the device, and its id. If the simulation is multithreaded or distributed, multiple files will be created, one for each process and/or thread. For more information on how to customise the behaviour and output format of recording devices, please read the documentation for `RecordingDevice`.

resetting simulations

It often occurs that we need to reset a simulation. For example, if you are developing a script, then you may need to run it from the `ipython` console multiple times before you are happy with its behaviour. In this case, it is useful to use the function `ResetKernel()`. This gets rid of all nodes you have created, any customised models you created, and resets the internal clock to 0.

The other main use of resetting is when you need to run a simulation in a loop, for example to test different parameter settings. In this case there is typically no need to throw out the whole network and create and connect everything, it is enough to re-parameterise the network. A good strategy here is to create and connect your network outside the loop, and then carry out the parametrisation, simulation and data collection steps within the loop. Here it is often helpful to call the function `ResetNetwork()` within each loop iteration. It resets all nodes to their default configuration and wipes the data from recording devices.

command overview

These are the new functions we introduced for the examples in this handout.

Getting and setting basic settings and parameters of NEST

- `GetKernelStatus(keys=none)`:

  Obtain parameters of the simulation kernel. Returns:

  - Parameter dictionary if called without argument
  - Single parameter value if called with single parameter name
  - List of parameter values if called with list of parameter names
  - Set parameters for the simulation kernel.

Models

- `GetDefaults(model)`:

  Return a dictionary with the default parameters of the given `model`, specified by a string.

- `SetDefaults(model, params)`:

  Set the default parameters of the given `model` to the values specified in the `params` dictionary.

- `CopyModel(existing, new, params=None)`:

  Create a `new` model by copying an `existing` one. Default parameters can be given as `params`, or else are taken from `existing`.

Simulation control

- `ResetKernel()`:

  Reset the simulation kernel. This will destroy the network as well as all custom models created with `CopyModel()`. The parameters of built-in models are reset to their defaults. Calling this function is equivalent to restarting NEST.

- `ResetNetwork()`:

  Reset all nodes and connections to the defaults of their respective model.