PCSIM Tutorial Exercises

FIAS Theoretical Neuroscience Summer School 2008

Dejan Pecevski
Institute for Theoretical Computer Science
Graz University of Technology
A-8010 Graz, Austria
{dejan@igi.tugraz.at }

August 7, 2008

Introduction

The exercises explain the most basic commands and classes in PCSIM and their usage, through some typical examples. They are split in multiple parts, with the subsequent parts modifying and extending the models in the previous parts. NOTE: Please copy the current script with a new name when you start a new part of the exercise, and preserve the script for each part, as you will need it afterwards.

At the end of this document, there is a short reference of PCSIM commands and classes with all information needed to perform the operations required in the steps of the exercises. In the exercises' steps there are many references to some PCSIM constructs without full explanation on how to use them, so please consult the short reference for when you need additional explanations and examples related to the particular class/command. The items in the reference are not in alphabetical order.

Exercise 1. Small Models Composed of Several Neurons

In the first part of the exercise you will create one leaky integrate-and-fire (LIF) neuron receiving spiking inputs from 10 input neurons firing a Poisson constant rate process, simulate the model, and plot the spikes and the membrane potential of the neuron.

In the second part you will examine how the firing rate of the LIF neuron depends on the firing rates of its input neurons.

Part 1. Construct and simulate the model

Step 1: Create a new file to edit the first part of exercise 1 (e.g. `ex1_part1.py`).

Step 2: Import the necessary Python packages for this exercise and create the empty PCSIM network object with the following code

```
from pypcsim import *
from numpy import *
from pylab import *

net = SingleThreadNetwork()
```

Step 3: Create 10 Poisson input neurons (see reference for class `PoissonInputNeuron` and `net.create` command). Set the firing rate for the input neurons to 5 Hz.

Step 4: Create one leaky integrate-and-fire (LIF) neuron `LifNeuron`.
Use the following parameters for the LIF neuron:

Parameter	Value	Description
Rm	$10^8\Omega$	membrane resistance R_m
Cm	$3 \cdot 10^{-10}$ F	membrane capacitance C_m
Vresting	-0.07 V	resting potential $V_{resting}$
Vreset	-0.07 V	reset potential V_{reset}
Vthresh	-0.06 V	threshold potential V_{thresh}
Vinit	-0.07 V	initial membrane potential V_{init}
Trefract	$5 \cdot 10^{-3}$ s	refractory period $T_{refract}$

Step 5: Connect each of the input neurons to the LIF neuron `LifNeuron` with a `StaticSpikingSynapse` synapse. The command `net.connect` creates a connection between two neurons.

The parameters of the synapses should be set to:

Parameter	Value	Description
`tau`	$5 \cdot 10^{-3}$ s	synapse time constant
`delay`	$1 \cdot 10^{-3}$ s	delay of the connection
`W`	$1 \cdot 10^{-10}$ A	synaptic weight

To iterate through the list of IDs of the input neurons, returned by `net.create`, we will need to create a Python loop. Below is an example for a simple for loop in Python. *Be careful*, in Python every code line that is within the loop has to be indented by <tab> from the head line of the loop.

```
# Iterates and prints the elements
# of a python sequence (list,tuple etc.).
for i in L:
        print i
```

See http://docs.python.org/tut/node5.html#firststeps, and http://docs.python.org/tut/node6.html for more info about loops in Python. More information on manipulation of lists in Python can be found at http://docs.python.org/tut/node5.html#lists

Step 6: Setup two recorders to record the spike times and the membrane potential of the neuron with `net.record`.

Step 7: Run the simulation for 1 second with `net.simulate`.

Step 8: Plot the spike and the membrane potential using the following matplotlib code (the spike list is assumed to be in the variable `recorded_spikes`, and the membrane potential in the variable `recorded_vm`):

```
# plot the spikes
figure(1, figsize = [8,3])
plot(recorded_spikes, zeros(len(recorded_spikes)), '|',
                              markersize=50)
xlim(0,1)

# plot the membrane potential
figure(2)

plot(arange(0,1,1e-4), recorded_vm)
xlim(0,1)
```

Step 9: Adjust the weight of the synapse so that the firing rate of the neuron will be around 10 Hz.

Step 10: Run the script with the command

```
$ ipython -pylab myscript.py
```

Part 2. Output Firing Rate / Input Firing Rate Dependency

The goal of part 2 of the exercise is to plot a curve for the dependency of the output firing rate of the LIF neuron in the previous experiment on the firing

rate of the input neurons. Run the experiment in the previous part for different values of the firing rate of input neurons in the range from 0 to 200 Hz (with a step of 1 Hz). For each run calculate the firing rate of the LIF neuron, and plot a curve depicting this dependency.

Step 1: To perform the experiment several times we will need to put the code from the creation of the network to the retrieval of the results within a Python FOR loop.

A list with the first N non-negative integers in Python can be constructed with `range(N)`. For example,

```
>>> L = range(10)
>>> print L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

To iterate through the values of a list you can use the Python example in the previous part.

Also change the simulation time to 10 seconds, for better estimation of the average firing rate of the LIF neuron.

Change the initialization of the the input rate in the constructor of the Poisson input neurons with the current rate variable within the loop.

Step 2: Add a code for the calculation of the firing rate of the LIF neuron at the end of the body of the loop. Calculate it as the number of spikes in the output divided by the simulation time. The length of a Python list can be retrieved with the function `len(L)`.

Step 3: In order to plot the curve, we need to preserve the output firing rates for the different runs in a Python list. Use the `L.append(element)` for that to append the values to the list. Creation of an empty list at the beginning can be done with `L = []`. More information on manipulation of lists in Python can be found at http://docs.python.org/tut/node5.html#lists

Step 4: Plot the curve with the following matplotlib code. The `rates` variable represents the list of the output firing rates. The **max_rate** is the maximum input firing rate for which the experiment was performed.

```
plot(range(max_rate), rates)
```

Exercise 2. Random Balanced Network with Excitatory and Inhibitory Populations

In this exercise you will construct a randomly connected recurrent neural network composed of 1000 LIF Neurons, where 80% of them are excitatory and 20% inhibitory. The neurons should be connected with current based synapses with exponentialy decaying synaptic response (type `StaticSpikingSynapse`), with random connection probability of $p = 0.1$. The input to the network will be composed of 100 Poisson input neurons (type `PoissonInputNeuron`) projecting random connections to the recurrent network with probability $p_{input} = 0.1$. After performing the simulation you will print the spike raster of the network activity, and some other number figures about the activity.

Part 1. Construct and Simulate the Random Network

Step 1: Create a new script for this exercise e.g. `exercise2.py`. At the beginning add several Python packages that will be necessary, and construct an empty PCSIM Network.

```
from pypcsim import *
from pypcsimplus import *
from numpy import *
from pylab import *

net = SingleThreadNetwork()
```

Step 2: Create two populations of LIF Neurons (type `LifNeuron`), one excitatory and one inhibitory with the appropriate number of neurons. See `SimObjectPopulation` class in the short PCSIM reference on how to do that.

Use the following parameters for the LIF neuron:

Parameter	Value	Description
Rm	$10^8\Omega$	membrane resistance R_m
Cm	$2 \cdot 10^{-10}$ F	membrane capacitance C_m
Vresting	-0.06 V	resting potential $V_{resting}$
Vreset	-0.06 V	reset potential V_{reset}
Vthresh	-0.05 V	threshold potential V_{thresh}
Vinit	-0.06 V	initial membrane potential V_{init}
Trefract	$3 \cdot 10^{-3}$ s	refractory period $T_{refract}$

Step 3: Construct a new population composed of the already created neurons in both populations. We will need this population for easier construction of the connections and setup of the recordings. Use list(`SimObjectPopulation.idVector()`)

statement to get a python list of the IDs of the objects in the excitatory and inhibitory populations, concatenate the two lists (in Python: `L = L1 + L2`), and then to create the new population use the constructor of `SimObjectPopulation` which accepts a list of IDs of already created objects in the network. See more info about the `SimObjectPopulation` class in the short PCSIM reference.

Step 4: Because we will use different parameters for the synapses connecting depending whether the synapse has a excitatory or inhibitory presynaptic neuron, the synaptic connections from the excitatory neurons to all neurons should be created with a separate projection (type `ConnectionsProjection`).

Make a projection from the excitatory population to the population with all neurons, with random connections (0.1 connectivity probability). You can find more info about `ConnectionsProjection` in the short PCSIM reference. Use the `StaticSpikingSynapse` synapse type with the following parameters

Parameter	Value	Description
`tau`	$5 \cdot 10^{-3}$ s	synapse time constant
`delay`	$1 \cdot 10^{-3}$ s	delay of the connection
`W`	$1.62 \cdot 10^{-11}$ A	synaptic weight

Step 5: Make a projection from the inhibitory population to the population with all neurons, with random connections (0.1 connectivity probability). Use the `StaticSpikingSynapse` synapse type with the following parameters

Parameter	Value	Description
`tau`	$10 \cdot 10^{-3}$ s	synapse time constant
`delay`	$1 \cdot 10^{-3}$ s	delay of the connection
`W`	$-10 \cdot 10^{-11}$ A	synaptic weight

Step 6: Create an input population of 100 Poisson input neurons (type `PoissonInputNeuron`) Connect the input population to the network with random connections (p = 0.1), and excitatory `StaticSpikingSynapses` synapse types with following parameters

Parameter	Value	Description
`tau`	$10 \cdot 10^{-3}$ s	synapse time constant
`delay`	$1 \cdot 10^{-3}$ s	delay of the connection
`W`	$1.2 \cdot 10^{-11}$ A	synaptic weight

Step 7: Create a recorder population for the recording of spiking times of all neurons in the network. Use the `SimObjectPopulation.record` method for that.

Step 8: Simulate the network for 1 second.

Step 9: Retrieve the spike times from all neurons. Create a list of numpy arrays, where each array holds the spike times of one neuron. The following code will perform that (`popul` is the variable of the neuron population) :

```
spikes = [ array(popul.object(i).getRecordedValues())
                    for i in range(popul.size()) ]
```

Step 10: To plot the spike raster we will use the function `create_raster` from the extra pcsim package pypcsimplus. This function creates list of x and y coordinates from the spike times and neuron IDs, which are then ready to be used in the plot command.

```
create_raster(spikes,0,Tsim)
plot(x, y, '.')
```

Step 11: To get familiar with the common operations with numpy arrays in Python, try to calculate the following figures from the spiking activity:

- The mean firing rate in the network
- The mean inter-spike interval for all neurons
- The mean coefficient of variation

Part 2. Random Distributions for Parameter Values

In this part you will modify the previous model by specifying random distributions that are to be used for the generation of parameter values during the creation of neurons within a population, or synapses within a projection. You will use the `SimObjectVariationFactory` to achieve this.

Step 1: Change the neuron model in the creation of the populations with a variation factory where the `Vinit` parameter of the neuron is set to a bounded normal distribution with mean $m = -55$ mV and deviation 0.1 times the mean, with bounds set to $[-60mV, -50mV]$. See the reference for `BndNormalDistribution` and `SimObjectVariationFactory` classes.

Step 2: Change the synapse model in the creation of the projections with a variation factory, and set the weights of the synapses to change according to a bounded gamma distribution with mean the previous constant value of the weight and standard deviation equal to 0.1 of the mean, within the interval between 0 and twice the mean value. See the reference for `BndGammaDistribution`.

Step 3: Run the two models, with and without randomly distributed parameters, and compare the spiking activity.

Part 3. Spatial Networks and Distance-based Random Connections

In this part you will further modify the model by introducing 3D coordinates for the neurons with the `SpatialFamilyPopulation` class. Then you will change the synaptic connections generation from random with fixed probability, to random with distance dependent probability, where the probability to make a connection between two neurons will depend on their distance according to the formula

$$p = C \cdot e^{-D^2/\lambda^2}$$

where D is the distance between the two neurons.

Step 1: Replace the creation of the populations in the previous model, by creating one `SpatialFamilyPopulation` where the neurons will be located on a 3D grid with integer coordinates, within a volume (20x10x5) (`CuboidIntegerGrid3D`). The population should have two families with the same neuron model used for both of them. The ratio of number of neurons in the two families should be specified with the `RatioBasedFamilies` class. Then split the population in two subpopulations corresponding to the two families (`SpatialFamilyPopulation.splitFamilies` method).

Step 2: Change the creation of the projections to use distance based random connections. Replace the `RandomConnections` connection iterator in the constructor of the projections, with the `EuclideanDistanceRandomConnections` class. Use the following parameters in the constructor of `EuclideanDistanceRandomConnections`: $C = 0.1$ and $\lambda = 10$.

Step 3: Run the model and plot the spiking activity.

Exercise 3. Models with Spike-Timing-Dependent Plasticity

In the following exercise you will modify the model from exercise 1, part 1 to include spike-timing-dependent plasticity in the synapses. You will also plot the evolution of the synaptic weight during the simulation of one of the synapses, and the histogram of the synaptic weight values at the beginning and at the end of the simulation.

Step 1: In the model replace the `StaticSpikingSynapse` synapse type with the `StaticStdpSynapse` type. Go to the following link in the PCSIM C++ class reference online for more info on this synapse model: click here
http://www.lsm.tugraz.at/pcsim/cppclassreference/html/classStaticStdpSynapse.html#_details
Use the following parameters for the synapse

Parameter	Value	Description
Winit	$1 \cdot 10^{-11}$ A	initial synaptic weight
tau	$5 \cdot 10^{-3}$ s	synaptic time constant
delay	$1 \cdot 10^{-3}$ s	conduction delay
useFroemkeDanSTDP	False	whether to use adjacent spike depression
Apos	$2 \cdot 10^{-13}$ A	Amplit. of the positive part of the STDP window
Aneg	$-2.2 \cdot 10^{-13}$ A	Amplit. of the negative part of the STDP window
taupos	$30 \cdot 10^{-3}$ s	time constant of the positive part of the STDP window
tauneg	$30 \cdot 10^{-3}$ s	time constant of the negative part of the STDP window
mupos	0.0	exponent for the STDP potentiation
muneg	0.0	exponent for the STDP depression
Wex	$2 \cdot 10^{-11}$ A	upper hard bound of the weights (2 times the initial weight)

Step 2: Change the number of input neurons to 100, and the simulation time to 100 seconds. Be sure that duration parameter of the `PoissonInputNeuron` input neurons is not smaller than the simulation time.

Step 3: Setup a `SimObjectPopulation` for the synapses. First create a python list of the IDs of the created synapses. net.connect returns the ID of the created synapse.

From this list of IDs create a `SimObjectPopulation` (see the short PCSIM reference).

Step 4: Then create a population of analog recorders to record the parameter "W" of the synapses. Since the synaptic weight is a typically slow changing value, we don't need to record the value at each time step, instead we will record each 1000 time steps (each 0.1 seconds). This is achieved by specifying `AnalogRecorder(samplingTime = 1000)` as a recorder.

Step 5: After the simulation create a list of numpy arrays of the recorded weight values, in the same way as for the spikes in the previous exercise.

```
weights = [ array(popul.object(i).getRecordedValues())
                    for i in range(popul.size()) ]
```

and then convert it to a two-dimensional numpy array

```
weights = vstack(weights)
```

Step 6: Plot the recorded weight of the first synapse (`weights[0]`).

Step 7: Plot the histogram of the weights at the beginning and at the end of the simulation, with the following code:

```
# plot a histogram of the weights at
# the beginning of the simulation
figure(2)
subplot(2,1,1)
hist(weights[:,0])

# plot a histogram of the weights at
# the end of the simulation
subplot(2,1,2)
hist(weights[:,-1])
```

Short reference to commonly used PCSIM commands and classes

- `net.create`

  Creates one or an array of PCSIM objects (e.g. neurons). Example:

```
# create one neuron, returns the ID of the
# created neuron
nrn1 = net.create( LifNeuron(Rm = 1e8))

# create 10 neurons, returns a list of neuron IDs
nrn2 = net.create( LifNeuron(Rm = 1e8), 10)
```

- `net.connect`

  Connects two neurons with a specified synapse type. Usage example:

```
# Connects the neuron with ID nrn1 to the neuron
# with ID nrn2 with a static current-based synapse
net.connect( nrn1, nrn2, StaticSpikingSynapse( tau = 3e-3 ))
```

- `net.record`

  Setup a recorder to record spikes or analog signals from a neuron/synapse (or other PCSIM object). Example:

```
# Create a recorder to record the spikes of a neuron.
# Returns the ID of the recorder.
spike_rec = net.record( nrn, SpikeTimeRecorder() )

# Create an analog recorder to record the membrane
# potential of the neuron. Returns the ID of the recorder.
vm_rec = net.record( nrn, "Vm", AnalogRecorder() )

# Record the weight of a plastic synapse
w_rec = net.record( syn, "W", AnalogRecorder() )
```

- `net.simulate`

  Simulates the network. Example:

```
# simulate for 3.5 seconds.
net.simulate(3.5)
```

- `LifNeuron`

  The description of the LIF neuron model can be found in the PCSIM C++ class reference documentation at http://www.lsm.tugraz.at/pcsim/cppclassreference/html/classLifNeuron.html#_details.

  You can also see the list of all parameters that you can set for the neuron, and their default values under the section *Constructor & Destructor documentation* below the description of the model.

- `PoissonInputNeuron`

  Input neuron which emits spikes obeying a homogenous Poisson process with a specific rate and duration. If you want the neuron to output spikes during the whole simulation then set the duration larger than the simulation time.

  Example:

```
# Create a poisson input neuron firing at rate 10 Hz for
# the duration of 10 seconds.
nrn = net.create( PoissonInputNeuron( rate = 10,
                                      duration = 10) )
```

- `StaticSpikingSynapse`

  The `StaticSpikingSynapse` class implements a current-based synapse with an exponentially decaying postsynaptic response, i.e. the dynamics of the current is defined with

$$\frac{di(t)}{dt} = -\frac{i(t)}{\tau_{syn}} + \sum_k w\delta(t - t^{(k)} - t_{delay})$$

  where t_k are the spike times of the presynaptic neuron. The parameters names in the constructor are `tau`, `W` and `delay`.

- `SpikeTimeRecorder`

  Recorder of spike times. Example:

```
# create a SpikeTimeRecorder and attach it to neuron nrn
rec = net.record(nrn,SpikeTimeRecorder())
# After simulation you obtain the
# recorded values in a python list with:
values = list(net.object(rec).getRecordedValues())
```

- `AnalogRecorder`

  Recorder of analog signals. Can be attached to an output analog port, or to a field of a PCSIM object. Example:

```
# create an AnalogRecorder and attach it to neuron
# nrn's field named "Vm" (membrane potential)
rec = net.record(nrn,"Vm", SpikeTimeRecorder())
# After simulation obtain the recorded values in a
# python list with:
values = list(net.object(rec).getRecordedValues())
```

- `SimObjectPopulation`

  Class representing a simple (array like) population of PCSIM objects (e.g. neurons). The objects in the population are identified by their index number, from 0 to population_size - 1.

- Creation of `SimObjectPopulation`

  A `SimObjectPopulation` can be created in two ways. The first way is by specifying an object model, or a factory (one type of factory is `SimObjectVariationFactory`), and a number of objects to be created. For example to create population of 100 Izhikevich neurons write:

```
nrn_model = IzhiNeuron(Rm = 1e8, Cm = 2e-10)
popul = SimObjectPopulation(net, nrn_model, 100)
```

  The other way to create a population is by specifying a Python list with IDs of already created objects in the PCSIM network. For example, if `list_nrns` is the list of IDs then

```
popul = SimObjectPopulation(net, list_nrns)
```

- `SimObjectPopulation.object`, Accessing the objects in a population

```
# Get the ID of the neuron from the population with index 10
nrn_id = popul[10]

#  The id can then be used to access the neuron object
nrn = net.object(nrn_id)

# Or alternatively get the neuron object directly
# from the population
nrn = popul.object(10)
# Then use the neuron like a regular python object
nrn.Vresting = -0.07
```

- `SimObjectPopulation.record`, Creating a recorder population

  In order to record spikes or analog signals from all the objects in a population, a population of recorders can be created corresponding to an existing object population. Each recorder is attached to one of the objects in the population.Example:

```
# create a population of recorders to record
# the spikes from the neurons in a population
rec_popul = popul.record( SpikeTimeRecorder() )

# create a population of recorders to record
# the membrane potential of the neurons in a population
rec_popul = popul.record( AnalogRecorder(), "Vm" )
```

- `SimObjectPopulation.size()`

  Returns the number of objects in the population

- `ConnectionsProjection`

  Construct to create a set of synaptic connections from a source population to a destination population (the source and destination can be the same). The rules for creating the connections are typicaly probabilistic and are specified by means of a `ConnectionIterator` class given in the constructor of the projection. Two commonly used connection iterators are `RandomConnections` and `EuclideanDistanceRandomConnections`.

  Creating a `ConnectionsProjection`

  Example:

```
synapse_model = StaticSpikingSynapse( w = 1e-10 )

# Create random connections with prob. = 0.2,
#from src_popul to dest_popul, by using previous
# the synapse model
proj = ConnectionsProjection( src_popul, dest_popul,
                                    synapse_model,
                                    RandomConnections( 0.2 ) )
```

- `BndNormalDistribution`

  A bounded normal random distribution, used commonly within `SimObjectVariationFactory` to associate normal distribution to parameter values. See `SimObjectVariationFactory` for more examples. If the generated random value does not fall within the bounds, then it's generated from an uniform distribution within these bounds.

  Example:

```
# create a normal distribution with
# mean 10, standard deviation 0.1 of its mean
# and with [0,20] upper and lower bounds.
dist = BndNormalDistribution( mu = 10, cv = 0.1,
                              lowerBound = 0,
                              upperBound = 20 )
```

- `BndGammaDistribution`

  A bounded gamma distribution, used commonly within `SimObjectVariationFactory` to associate gamma distribution to parameter values. See `SimObjectVariationFactory` for more examples. If the generated random value is larger than the upper bound, then it's generated from an uniform distribution within 0 and the upper bound.

  Example:

```
# create a gamma distribution with
# mean 10, standard deviation 0.1 of its mean
# and with 20 upper bound.
dist = BndNormalDistribution( mu = 10, cv = 0.1,
                              upperBound = 20 )
```

- `SpatialFamilyPopulation`

Heterogenous population of objects where each PCSIM object has associated a 3D coordinate in space and a family ID. The created population is a heterogenous population, which means that multiple PCSIM object factories can be specified for the generation of the objects within the population. The different object factories can populate the population with different object types. The PCSIM objects generated from one object factory are refered to as one family of PCSIM objects. A special `SpatialFamilyIDGenerator` object used in the constructor of the population chooses among the object factories to create an object for each location in space. One commonly used `SpatialFamilyIDGenerator` is `RatioBasedFamilies`. The set of 3D locations are specified with a `Point3DSet`. Commonly used `Point3DSet` is the `CuboidIntegerGrid3D`.

More info on the `SpatialFamilyPopulation` can be found here (http://www.lsm.tugraz.at/pcsim/cppclassreference/html/classSpatialFamilyPopulation.html#_details)

Example: Creates a population with two families of neurons, the first of type `LifNeuron` and the second of type IzhiNeuron. The neurons are located on a cuboid 3d grid with integer coordinates, of dimensions 10x5x5, and the origin at (0,0,0). During the generation of the neurons, the probability for a neuron from the first family to be created is 0.8 (4/5), and from the second family 0.2 (1/5). This gives an average ratio of 4:1 for the number of the neurons in the two families.

```
popul = SpatialFamilyPopulation( net,
              [ LifNeuron(Rm = 1e8), IzhiNeuron(Rm = 1e7) ],
               RatioBasedFamilies((4,1)),
               CuboidIntegerGrid3D(10,5,5))
```

- `SimObjectVariationFactory`

  Used to create objects in a population where some of the parameter values of the objects are drawn from a random distribution during generation.

  Example: The values of the conduction delay parameter of the connections created in the projection are drawn from a normal distribution.

```
# first create the synapse model
syn_model = StaticSpikingSynapse(W = 1e-10,
                         tau = 3e-3, delay = 1e-3)

# Then create a variation factory based on the model
syn_factory = SimObjectVariationFactory( syn_model )

# create the random distribution
dist = BndNormalDistribution( mu = 2e-3, cv = 0.3,
                              lowerBound = 1e-3 ,
                              upperBound = 8e-3 )
# associate the delay parameter of the synapse model
# with the random distribution
syn_factory.set( "delay", dist )

# create the projection using the factory
# to create the synapses
proj = ConnectionsProjection(src_popul, dest_popul,
                 syn_factory, RandomConnections( 0.01 ) )
```

- `EuclideanDistanceRandomConnections`

  A specific `ConectionIterator` class used in a projection to create random connections with distance dependent probability, i.e. the probability to make a connection between two neurons depends on the distance between them according to the formula

  $$p = C \cdot e^{-D^2/\lambda^2}$$

  where D is the distance between the two neurons.

  Example:

```
con_iterator =
EuclideanDistanceRandomConnections( C = 0.1,
                                    lambda = 3 )

proj = ConnectionsProjection( pop1, pop2,
                              synapse_model,
                              con_iterator )
```

- `RandomConnections`

A specific `ConectionIterator` class used in a projection to create random connections with fixed probability. See `ConnectionsProjection` for an usage example.