# FYS388_V15_NEST_By_Example

March 24, 2015

# 1 NEST by Example: An Introduction to the Neural Simulation Tool NEST

#### 1.0.1 Marc-Oliver Gewaltig, Abigail Morrison, Hans Ekkehard Plesser

#### 1.0.2 This notebook adapted to NEST 2.6.0 was created by Hans Ekkehard Plesser

## 1.1 About this IPython Notebook

This notebook is based on the tutorial *NEST by Example: An Introduction to the Neural Simulation Tool NEST*, by Marc-Oliver Gewaltig, Abigail Morrison, and Hans Ekkehard Plesser. The first version of the tutorial, covering NEST 1.9.8194, was published as Chapter 18 of *Computational Systems Neurobiology* edited by Nicolas Le Novère, Springer, 2012.

The document is updated with every new release of NEST. You can find the current version on the NEST Homepage and in the NEST source code, directory `doc/nest_by_example`, including all example scripts.

For a thorough explanation of the code in this notebook, please refer to the Tutorial.

### 1.1.1 Installing NEST

- Please see the NEST Installation Instructions for information on how to install NEST.
- To work with this Notebook, you need to build NEST with Python (this is the default).
- You should build NEST without MPI support to begin with (this is the default), because the NEST help system does not work fully when NEST is built with MPI.
- The easiest way to get started may be to download our Ubuntu Live Media with NEST 2.6.0 image and run it as a virtual machine on your computer using Virtual Box.

## 1.2 Basic Terminology

- **Nodes** Nodes are all neurons, devices, and also sub-networks. Nodes have a dynamic state that changes over time and that can be influenced by incoming events. In NEST, we can distinguish roughly between three types of nodes
  - **Simulation devices** such as `poisson_generator`, `spike_generator` or `noise_generator`. These devices only produce output (spikes or currents) that is sent to other nodes, but do not receive any input.
  - **Recording devices**, such as `spike_detector`, `voltmeter` or `multimeter`. These devices collect data from other nodes and store it in memory or in files for analysis. They only receive input, but do not emit any output events to other nodes.
  - **Neurons**, such as `iaf_neuron` or `iaf_psc_alpha` represent our model neurons in the network. They can receive spike and current input and emit spike output.
- **Events** Events are pieces of information of a particular type. The most common event is the spike-event. Other event types are voltage events and current events.

- **Connections** Connections are communication channels between nodes. Only if one node is connected to another node, can they exchange events. Connections are weighted, directed, and specific to one event type.
  - **Source**: The node that sending an event (spike)
  - **Target**: The node that receiving an event (spike)
  - **Weight**: How strongly will an event will influence the target node?
  - **Delay**: How long does the event take from source to target?

## 1.3 Notebook preparations

We first import some packages that we will need for data analysis and plotting (NumPy, Pandas, Matplotlib) and instruct the notebook to render all figures inside the notebook. We also increase the figures size, as the default figure size is rather small for most modern displays.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from IPython.display import Image, SVG

        %matplotlib inline
        plt.rcParams['figure.figsize'] = (12, 6)      # this line must come after the %matplotlib inline
```

## 1.4 Firing up NEST

1. Try `import nest` first. If that works, proceed to **Getting Help**.
2. If that fails, complaining that it cannot find module `nest`, you need to extend you Python path to include the installation location of the NEST Python module on your computer.
3. Adapt the code in the next cell to your installation location.

```
In [2]: import sys
        sys.path.append('/Users/plesser/NEST/code/releases/nest-2.6.0/ins/lib/python2.7/site-packages/')
```

```
In [3]: import nest
        import nest.voltage_trace
        import nest.raster_plot
```

## 1.5 Getting Help

NEST provides several levels of online documentation

**Help on the NEST Python interface** To obtain help on the NEST Python interface, you can either use the Python `help()` function or the IPython help mechanism by placing a `?` behind a command name, e.g.,

```
In [4]: help('nest.Create')
```

```
Help on function Create in nest:

nest.Create = Create(*args, **kwargs)
    Create n instances of type model. Parameters for the new nodes can
    are given as params (a single dictionary or a list of dictionaries
    with size n). If omitted, the model's defaults are used.
```

```
In [5]: nest.Create?
```

Executing the cell above will show the help at the bottom of the IPython window.

**The NEST Helpdesk** NEST has an online helpdesk. Unfortunately, the `nest.helpdesk()` command is broken in most situations. The following line will print out the address of the helpdesk. Paste it into your favourite browser!

```
In [6]: print nest.sli_func('statusdict/prgdocdir :: (/index.html) join')
```

```
/Users/plesser/NEST/code/releases/nest-2.6.0/ins/share/doc/nest/index.html
```

## 1.6 A first example

- Single leaky integrate-and-fire neuron
- Sinusoidal current injection (`ac_generator`)
- Excitatory and inhibtory spike input (`poisson_generator`) via current-based synapses
- Record voltage trace with `voltmeter`

#### Reset NEST to its original state

```
In [7]: nest.ResetKernel()
```

#### Create the network

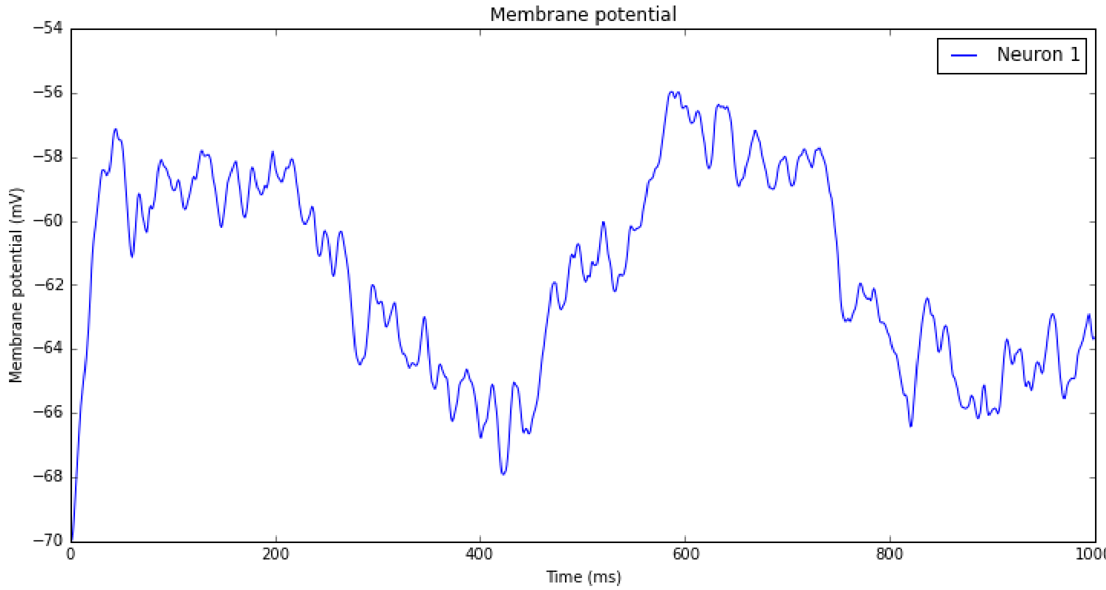
```
In [8]: neuron = nest.Create('iaf_neuron')

        sine = nest.Create('ac_generator', 1,
                           {'amplitude': 100.0, 'frequency': 2.0})

        noise = nest.Create('poisson_generator', 2,
                            [{'rate': 70000.0}, {'rate': 20000.0}])

        voltmeter = nest.Create('voltmeter', 1, {'withgid': True})

        nest.Connect(sine, neuron)
        nest.Connect(voltmeter, neuron)
        nest.Connect(noise[:1], neuron, syn_spec={'weight': 2.0, 'delay': 1.0})
        nest.Connect(noise[1:], neuron, syn_spec={'weight': -5.0, 'delay': 1.0})
```

#### Simulate the network

```
In [9]: nest.Simulate(1000.0)
```

#### Visualize the results

```
In [10]: nest.voltage_trace.from_device(voltmeter);
```




### 1.6.1 Showing the network structure (not connections)

```
In [11]: nest.PrintNetwork()
```

Due to limitations in the IPython Notebook interface, the output is not shown here, but in the Terminal window in which you started `ipython notebook`. I glue it in here:

```
+-[0] root dim=[5]
   |
   +-[1] iaf_neuron
   +-[2] ac_generator
   +-[3]...[4] poisson_generator
   +-[5] voltmeter
```

### 1.6.2 Inspecting the connections

`GetConnections()` returns a list with one element per connection, each element having the following structure 1. GID of sender 1. GID of target 1. Target thread 1. Synapse-type id 1. Port number

```
In [12]: conns = nest.GetConnections()
         conns

Out[12]: (array('l', [2, 1, 0, 0, 0]),
          array('l', [3, 1, 0, 0, 0]),
          array('l', [4, 1, 0, 0, 0]),
          array('l', [5, 1, 0, 0, 0]))
```

To find out more about the connection properties, we use `GetStatus()`

```
In [13]: for syn in nest.GetStatus(conns):
             print """
                   Source: {source}
                   Target: {target}
```

```
                 Model : {synapse_model}
                 Weight: {weight} pA
                 Delay : {delay} ms""".format(**syn)
```

```
Source: 2
          Target: 1
          Model : static_synapse
          Weight: 1.0 pA
          Delay : 1.0 ms

          Source: 3
          Target: 1
          Model : static_synapse
          Weight: 2.0 pA
          Delay : 1.0 ms

          Source: 4
          Target: 1
          Model : static_synapse
          Weight: -5.0 pA
          Delay : 1.0 ms

          Source: 5
          Target: 1
          Model : static_synapse
          Weight: 1.0 pA
          Delay : 1.0 ms
```

### 1.6.3 Using Pandas with NEST

- Pandas is a powerful tool for managing and analysing large amounts of data
- Data returned by NEST can in many cases be easily converted to Pandas dataframes
- To first approximation, Pandas dataframes work like NumPy arrays with named columns

#### Extracting connection data

- We use the same `GetStatus()` call as above
- We obtain a tuple with one dictionary per connection
- We can convert this directly into a dataframe
- If we `print` the data, we get an ASCII-pretty-printed table

```
In [14]: conn_data = pd.DataFrame.from_records(nest.GetStatus(conns))
         print conn_data
```

```
delay  receptor  sizeof  source   synapse_model  target  weight
0      1         0       32       2  static_synapse       1       1
1      1         0       32       3  static_synapse       1       2
2      1         0       32       4  static_synapse       1      -5
3      1         1       32       5  static_synapse       1       1
```

- If we just display the dataframe in the Notebook, we get a nicely formatted table

```
In [15]: conn_data
```

```
Out[15]:    delay  receptor  sizeof  source   synapse_model  target  weight
         0      1         0       32       2  static_synapse       1       1
```

```
1      1          0      32        3  static_synapse       1       2
2      1          0      32        4  static_synapse       1      -5
3      1          1      32        5  static_synapse       1       1
```

- We can even get LaTeXcode

```
In [16]: print conn_data.to_latex()
```

```
\begin{tabular}{lrrrrlrr}
\toprule
{} &  delay &  receptor &  sizeof &  source &   synapse\_model &  target &  weight \\
\midrule
0 &      1 &         0 &      32 &       2 &  static\_synapse &       1 &       1 \\
1 &      1 &         0 &      32 &       3 &  static\_synapse &       1 &       2 \\
2 &      1 &         0 &      32 &       4 &  static\_synapse &       1 &      -5 \\
3 &      1 &         1 &      32 &       5 &  static\_synapse &       1 &       1 \\
\bottomrule
\end{tabular}
```

- By default, we get all properties of the connections
- We can also explicitly select only certain properties

```
In [17]: properties = ('source', 'target', 'delay', 'weight')
         conn_data = pd.DataFrame.from_records(list(nest.GetStatus(conns, keys=properties)), columns=pro

         conn_data
```

```
Out[17]:    source  target  delay  weight
         0       2       1      1       1
         1       3       1      1       2
         2       4       1      1      -5
         3       5       1      1       1
```

**Plotting "manually"**

- We can also extract data from recording devices into dataframes
- `voltmeter` is a single-element list (NEST commands take and return lists/tuples)
- Recorded data is in the `events` property of the voltmeter
- Data is recorded as a single-element tuple, we need to extract that element

```
In [18]: vm_data = pd.DataFrame(nest.GetStatus(voltmeter, 'events')[0])

         vm_data[:5]
```

```
Out[18]:          V_m  senders  times
         0 -70.000000        1      1
         1 -69.963834        1      2
         2 -69.746778        1      3
         3 -69.325646        1      4
         4 -68.819191        1      5
```

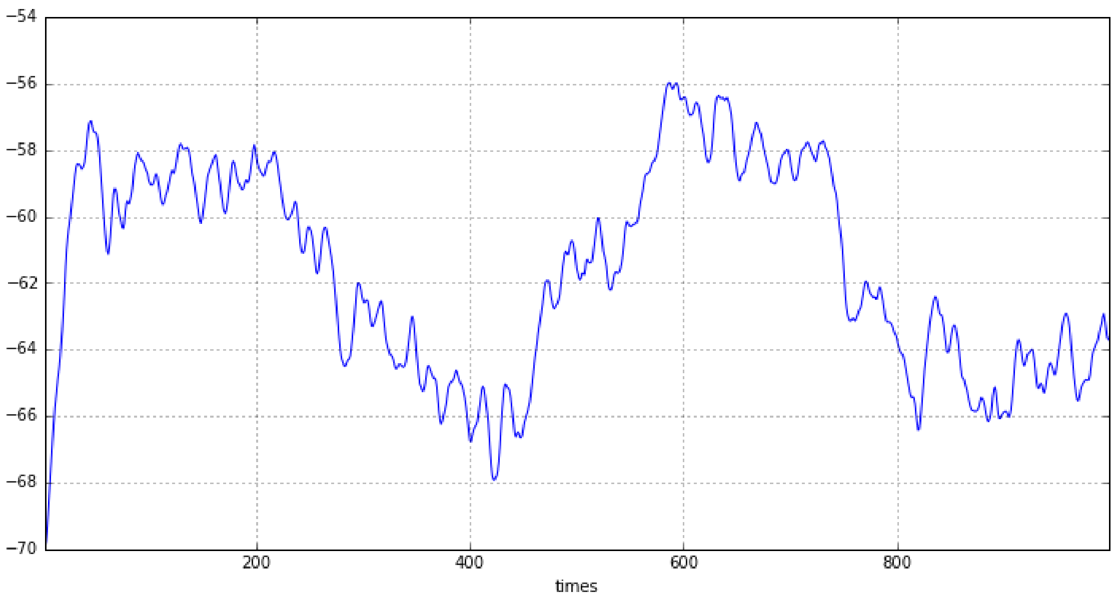
- We can obtain statistics on the data (here, meaningful for $V_m$ only)

```
In [19]: vm_data.describe()
```

```
Out[19]:              V_m  senders        times
         count  999.000000      999  999.000000
         mean   -61.607877        1  500.000000
         std      3.100941        0  288.530761
         min    -70.000000        1    1.000000
         25%    -64.393993        1  250.500000
         50%    -61.701118        1  500.000000
         75%    -58.701634        1  749.500000
         max    -55.968418        1  999.000000
```

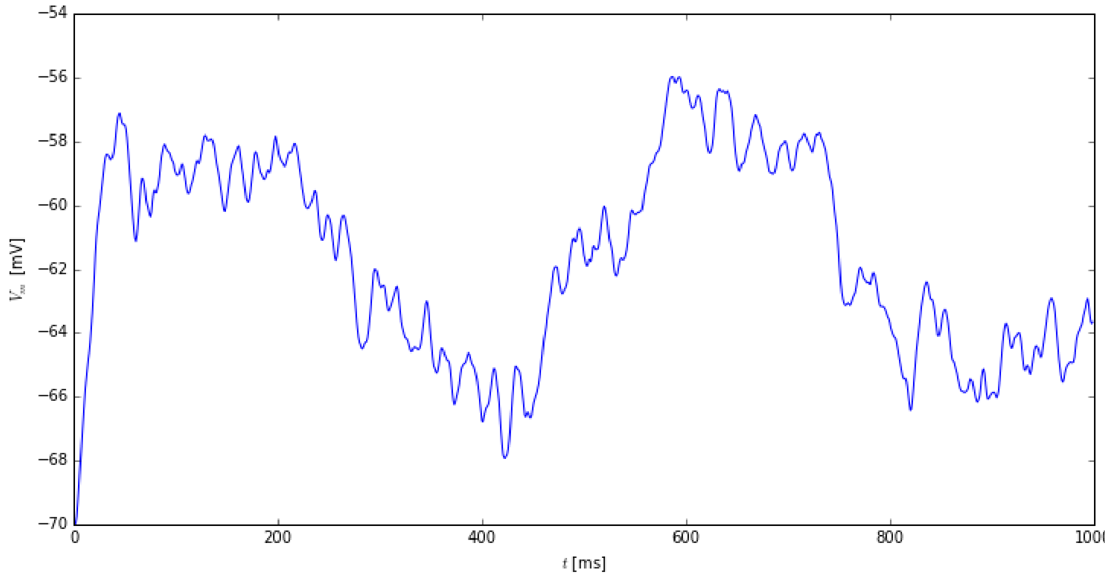
- We can plot using the `plot()` method of the Pandas DataFrame object (more on plotting in Pandas)

```
In [20]: vm_data.plot(x='times', y='V_m');
```




- We can plot using the normal `plot()` command

```
In [21]: plt.plot(vm_data.times, vm_data.V_m)
         plt.xlabel('$t$ [ms]')
         plt.ylabel('$V_m$ [mV]');
```

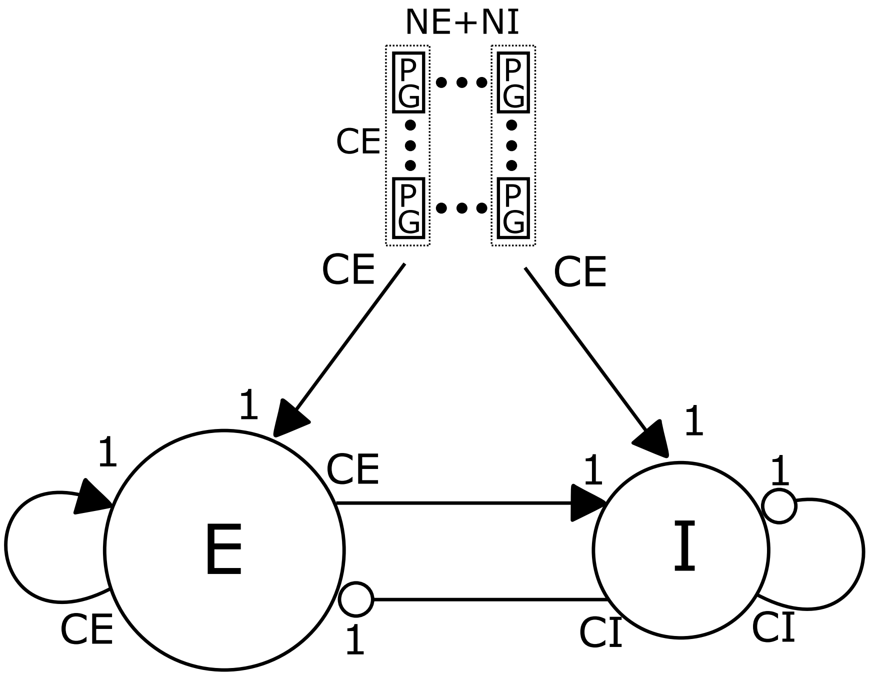



## 1.7 The Brunel Network: A sparsely connected network

- $N_E = 8000$ excitatory neurons
- $N_I = 2000$ inhibitory neurons
- 10% connectivity
- Scaling parameter allows adjustment of network size for faster testing (changes dynamics)
- N. Brunel, J Comput Neurosci 8:183 (2000)

In [22]:
```
SVG(filename='figures/brunel_detailed_external_single2.svg')
```

Out[22]:




### 1.7.1 Define parameters

```
In [23]: nest.ResetKernel()

         # Network parameters. These are given in Brunel (2000) J.Comp.Neuro.
         g       = 5.0    # Ratio of IPSP to EPSP amplitude: J_I/J_E
         eta     = 2.0    # rate of external population in multiples of threshold rate
         delay   = 1.5    # synaptic delay in ms
         tau_m   = 20.0   # Membrane time constant in mV
         V_th    = 20.0   # Spike threshold in mV

         scale = 1.
         N_E = int(scale * 8000)
         N_I = int(scale * 2000)
         N_neurons = N_E + N_I

         C_E    = int(N_E/10) # number of excitatory synapses per neuron
         C_I    = int(N_I/10) # number of inhibitory synapses per neuron

         J_E  = 0.5
         J_I  = -g*J_E

         nu_ex  = eta* V_th/(J_E*C_E*tau_m) # rate of an external neuron in ms^-1
         p_rate = 1000.0*nu_ex*C_E          # rate of the external population in s^-1
```

### 1.7.2 Configure kernel and neuron defaults

```
In [24]: nest.SetKernelStatus({"print_time": True,
                               "local_num_threads": 2})

         nest.SetDefaults("iaf_psc_delta",
                          {"C_m": 1.0,
                           "tau_m": tau_m,
                           "t_ref": 2.0,
                           "E_L": 0.0,
                           "V_th": V_th,
                           "V_reset": 10.0})
```

### 1.7.3 Create neurons

```
In [25]: nodes   = nest.Create("iaf_psc_delta", N_neurons)
         nodes_E = nodes[:N_E]
         nodes_I = nodes[N_E:]
```

### 1.7.4 Connect neurons with each other

```
In [26]: nest.CopyModel("static_synapse", "excitatory",
                        {"weight":J_E, "delay":delay})

         nest.Connect(nodes_E, nodes,
                      {"rule": 'fixed_indegree', "indegree": C_E},
                      "excitatory")

         nest.CopyModel("static_synapse", "inhibitory",
                        {"weight":J_I, "delay":delay})

         nest.Connect(nodes_I, nodes,
                      {"rule": 'fixed_indegree', "indegree": C_I},
                      "inhibitory")
```

### 1.7.5 Add stimulation and recording devices

```
In [27]: noise = nest.Create("poisson_generator", params={"rate": p_rate})

         # connect using all_to_all: one noise generator to all neurons
         nest.Connect(noise, nodes, syn_spec="excitatory")

         spikes=nest.Create("spike_detector", 2,
                            params=[{"label": "Exc", "to_file": False},
                                    {"label": "Inh", "to_file": False}])
         spikes_E = spikes[:1]
         spikes_I = spikes[1:]

         N_rec   = 100   # Number of neurons to record from

         # connect using all_to_all: all recorded excitatory neurons to one detector
         nest.Connect(nodes_E[:N_rec], spikes_E)
         nest.Connect(nodes_I[:N_rec], spikes_I)
```

### 1.7.6 Simulate

```
In [28]: simtime = 1000
         nest.Simulate(simtime)
```

### 1.7.7 Extract recoded data and display

#### Computing spike rates

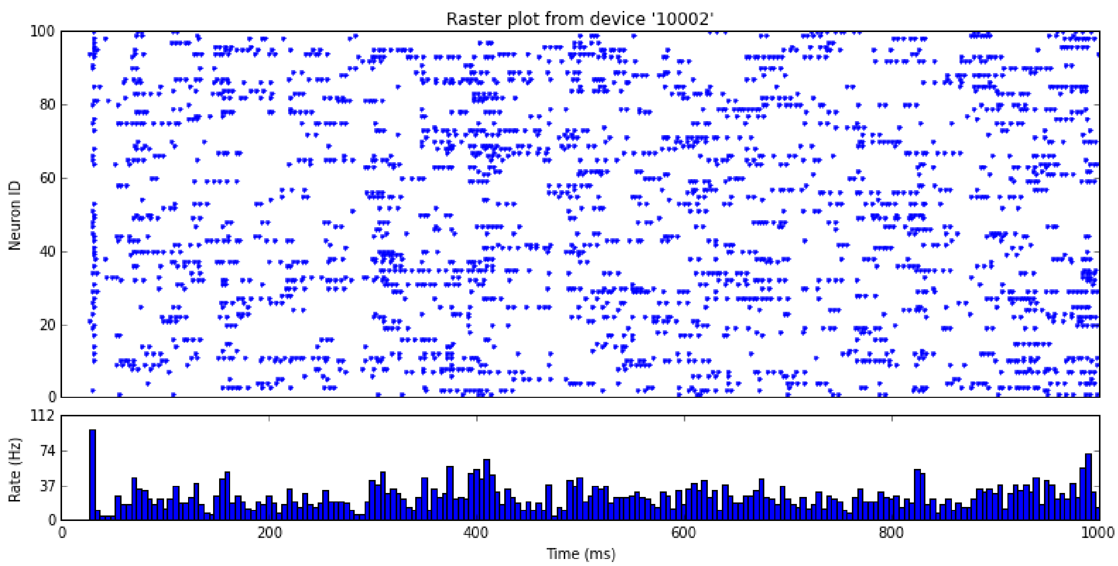
```
In [29]: events = nest.GetStatus(spikes, "n_events")

         rate_ex= events[0]/simtime*1000.0/N_rec
         print("Excitatory rate   : %.2f Hz" % rate_ex)

         rate_in= events[1]/simtime*1000.0/N_rec
         print("Inhibitory rate   : %.2f Hz" % rate_in)
```

```
Excitatory rate   : 20.00 Hz
Inhibitory rate   : 20.00 Hz
```

- **PROBLEM: Why are both rates *exactly* 20.00 Hz?**
- Check the code and spot the problem!

#### Visualization with raster_plot

```
In [30]: nest.raster_plot.from_device(spikes_E, hist=True);
```




#### Managing data with Pandas and plotting "manually"

```
In [31]: sdE_data = pd.DataFrame(nest.GetStatus(spikes_E, 'events')[0])
         sdI_data = pd.DataFrame(nest.GetStatus(spikes_I, 'events')[0])
```

```
In [32]: sdE_data[:5].T
```

Out[32]:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| senders | 94.0 | 90.0 | 26.0 | 2.0 | 80.0 |
| times | 26.8 | 29.4 | 29.6 | 29.8 | 29.8 |

In [33]:
```
sdE_data.describe().T
```

Out[33]:

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| senders | 2477 | 50.755753 | 29.610062 | 1.0 | 26.0 | 50 | 78 | 100.0 |
| times | 2477 | 529.420186 | 284.123144 | 26.8 | 307.4 | 522 | 781 | 998.7 |

In [34]:
```
sdI_data.describe().T
```

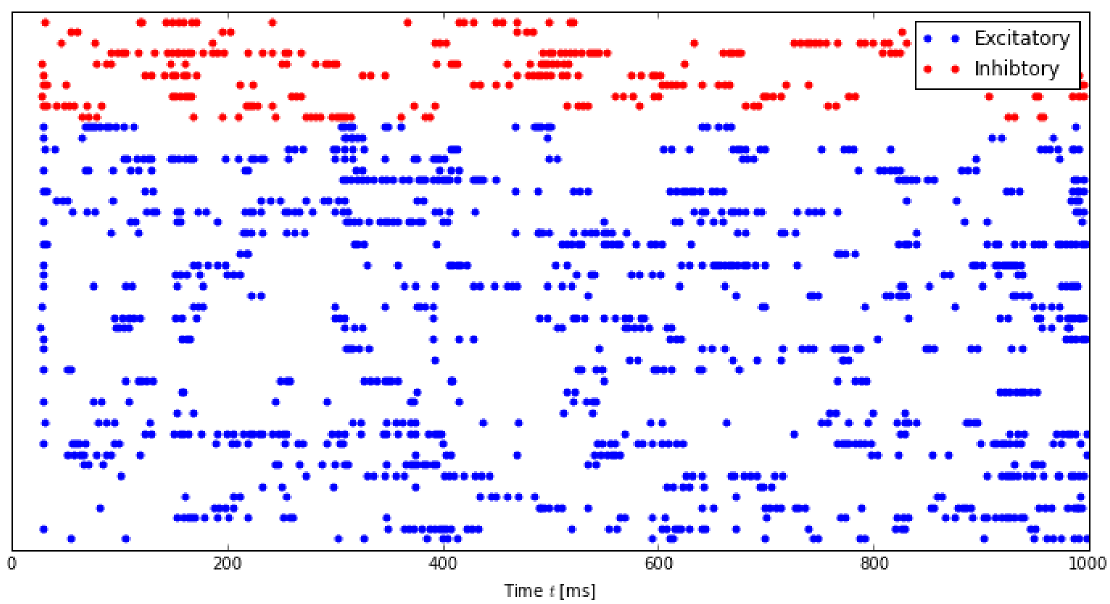
Out[34]:

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| senders | 2522 | 8050.627280 | 28.666002 | 8001.0 | 8027.000 | 8051.0 | 8075.00 | 8100 |
| times | 2522 | 533.317288 | 286.693879 | 27.4 | 300.025 | 527.7 | 799.35 | 998 |

**Raster plot of spikes**

- We plot sender GIDs vs spike times
- We only plot the first 40 excitatory neurons and the first 10 inhibitory neurons
- We need to offset the GIDs of the inhibitory neurons properly

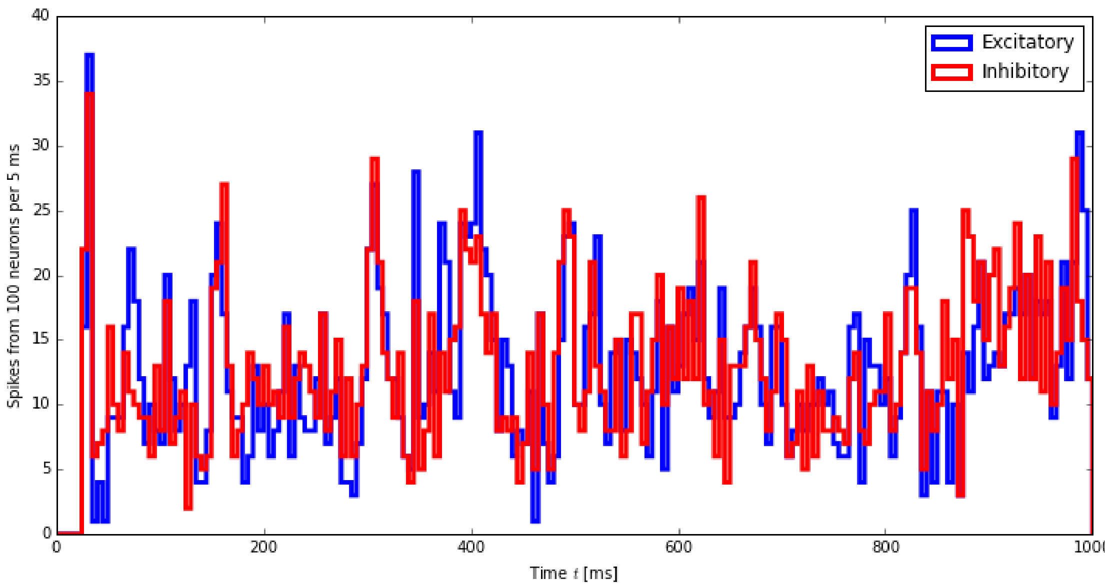
In [35]:
```
e_spikes_plot = sdE_data[sdE_data.senders <= 40]
i_spikes_plot = sdI_data[(8000 < sdI_data.senders) & (sdI_data.senders <= 8010)]

plt.plot(e_spikes_plot.times, e_spikes_plot.senders, 'bo', markersize=5, markeredgecolor='none
plt.plot(i_spikes_plot.times, i_spikes_plot.senders-8000+40, 'ro', markersize=5, markeredgecol
plt.legend()
plt.xlabel('Time $t$ [ms]')
plt.yticks([])
plt.ylim(0, 51);
```



#### Spike-time histogram

- Explicit bins give us better control over binning
- We could also use NumPy's histogram function
- Histogram type `step` usually gives cleaner diagrams when we have many bins

In [36]:
```
bins = np.arange(0., 1001., 5.)
plt.hist(sdE_data.times, bins=bins, histtype='step', lw=3, color='b', label='Excitatory')
plt.hist(sdI_data.times, bins=bins, histtype='step', lw=3, color='r', label='Inhibitory')
plt.legend()
plt.xlabel('Time $t$ [ms]')
plt.ylabel('Spikes from 100 neurons per 5 ms');
```



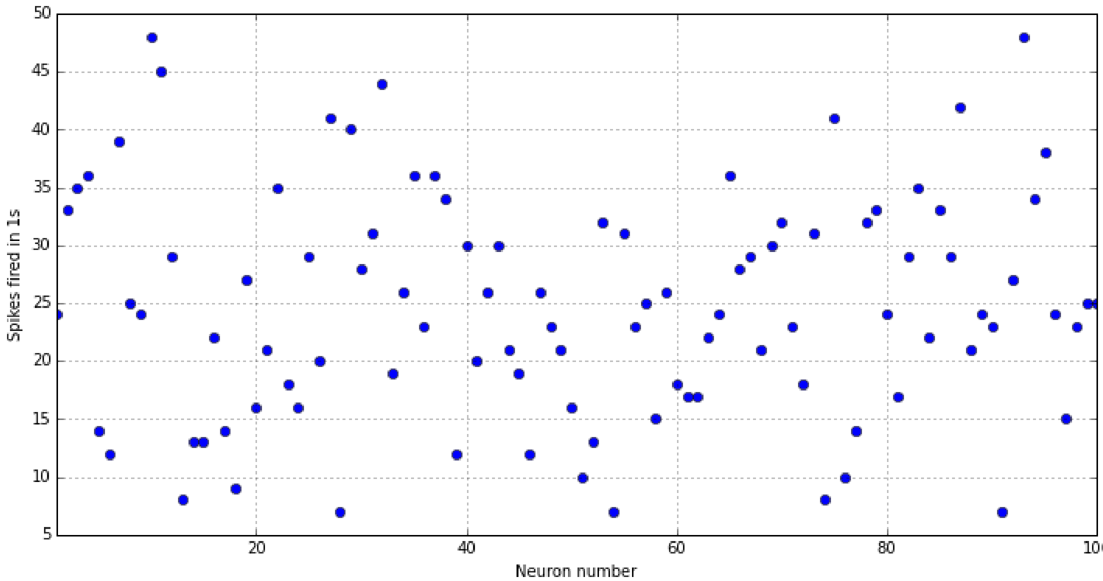

#### Interspike-interval histograms

- We need to compute ISIs for each neuron separately
- We therefore first group the data by senders (we focus on excitatory spikes here)
- See also the Pandas Groupby Documentation

In [37]:
```
e_spk_grouped = sdE_data.groupby('senders')
```

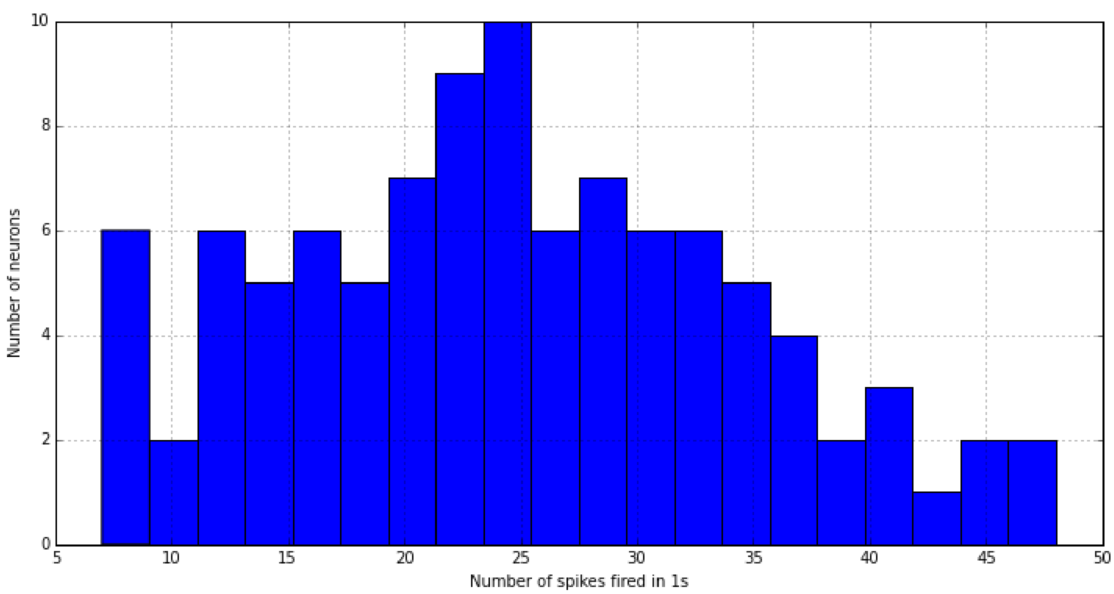
- As a first step, let us look at the firing activity of the individual neurons

In [38]:
```
e_spk_grouped.size().plot(style='bo')
plt.xlabel('Neuron number')
plt.ylabel('Spikes fired in 1s');
```




- Now, let us histogram this data

```
In [39]: e_spk_grouped.size().hist(bins=20)
         plt.xlabel('Number of spikes fired in 1s')
         plt.ylabel('Number of neurons');
```

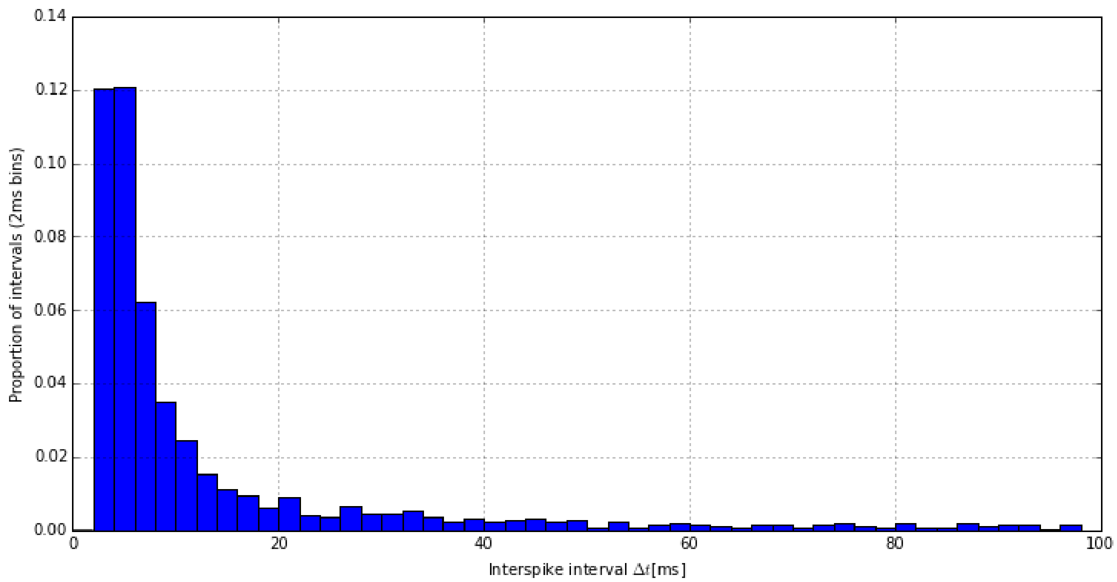



- We can now compute differences in spike times
- For the first spike of each neuron, we get a 'NaN' value, which we drop
- We thus get a data series of inter-spike interval

```
In [40]: e_spk_isi = e_spk_grouped['times'].diff().dropna()
```

- From that, we obtain the ISI histogram

```
In [41]: e_spk_isi.hist(bins=np.arange(0., 100., 2.), normed=True)
         plt.xlabel('Interspike interval $\Delta t$[ms]')
         plt.ylabel('Proportion of intervals (2ms bins)');
```




## 1.8 Randomizing Neurons and Synapses

### 1.8.1 Parallelism in NEST

- NEST can combine MPI-based and thread-based parallelism
- The key concept in parallel NEST is the *virtual process* (VP)
  - For $N_M$ MPI processes with $N_T$ threads each, be have

$$N_{VP} = N_M \times N_T$$

    virtual processes
  - For fixed $N_{VP}$, a NEST simulation shall yield identical results, independent of how the virtual processes are divided into MPI processes and threads
  - Recording devices write one file per MPI process, data from those files have to be pooled
- In this notebook, we only use thread-based parallelism: $N_{VP} = N_T$, $N_M = 1$

### 1.8.2 Random numbers in NEST

Random numbers may be drawn in three ways in a NEST simulation

1. globally in NEST, e.g., when creating divergent connections;
2. in NEST parallel on all VPs, e.g., when creating convergent connections;
3. in the Python script, e.g., when passing randomized parameters to NEST.

**Seeding random numbers**

- We thus have $N_{VP}+1$ random number generators in NEST, and need to supply seeds for all.
- In general, we also need one Python RNG per VP, see Tutorial Sections 5 and 6.
- Since we use only a single MPI process, we need only a single Python random number generator.
- In total, we need $N_{VP}+2$ seeds
- We build our seeds from a master seed
- **NOTE**: For different simulations, you need to use master seeds at least $N_{VP}+2$ apart

### 1.8.3 The Code

**Preparations first**

In [42]:
```
nest.ResetKernel()

# Number of virtual processes
n_vp = 4

# Master seed for simulation
master_seed = 12345

# Network parameters. These are given in Brunel (2000) J.Comp.Neuro.
g       = 5.0    # Ratio of IPSP to EPSP amplitude: J_I/J_E
eta     = 2.0    # rate of external population in multiples of threshold rate
delay   = 1.5    # synaptic delay in ms
tau_m   = 20.0   # Membrane time constant in mV
V_th    = 20.0   # Spike threshold in mV

scale = 1.
N_E = int(scale * 8000)
N_I = int(scale * 2000)
N_neurons = N_E + N_I

C_E    = int(N_E/10) # number of excitatory synapses per neuron
C_I    = int(N_I/10) # number of inhibitory synapses per neuron

J_E  = 0.5
J_I  = -g*J_E

nu_ex  = eta* V_th/(J_E*C_E*tau_m) # rate of an external neuron in ms^-1
p_rate = 1000.0*nu_ex*C_E          # rate of the external population in s^-1

# Limits for randomization
V_min, V_max = -V_th, V_th
w_min, w_max = 0.5 * J_E, 1.5 * J_E
```

**Configure kernel, including seeding RNGs**

In [43]:
```
seeds = range(master_seed, master_seed + n_vp + 2)

pyrng = np.random.RandomState(seeds[0])

nest.SetKernelStatus({'print_time': True,
                      'local_num_threads': n_vp,
                      'grng_seed': seeds[1],
```

```
                                'rng_seeds': seeds[2:]})

nest.SetDefaults("iaf_psc_delta",
                 {"C_m": 1.0,
                  "tau_m": tau_m,
                  "t_ref": 2.0,
                  "E_L": 0.0,
                  "V_th": V_th,
                  "V_reset": 10.0})
```

#### Creating neurons with randomized membrane potential

- We need to create the neurons first, then randomize $V_m$ uniformly over $[V_{\min}, V_{\max})$
- We use a variant of `SetStatus()` that allows us to pass one value per node
- **Note**: This code is inefficient when using many MPI processes

```
In [44]: nodes   = nest.Create("iaf_psc_delta", N_neurons)
         nodes_E = nodes[:N_E]
         nodes_I = nodes[N_E:]

         nest.SetStatus(nodes, params='V_m',
                        val=pyrng.uniform(low=V_min, high=V_max, size=(N_neurons,)))
```

#### Connecting with randomized weights

- We exploit that `Connect()` can draw random weights
- We only randomize outgoing connections from the excitatory neurons
- Random numbers are drawn from NEST's per-VP RNGs

```
In [45]: nest.CopyModel("static_synapse", "excitatory",
                        {"weight":J_E, "delay":delay})

         nest.Connect(nodes_E, nodes,
                      {"rule": 'fixed_indegree', "indegree": C_E},
                      {"model": "excitatory",
                       "weight": {"distribution": "uniform",
                                  "low": w_min,
                                  "high": w_max}})

         nest.CopyModel("static_synapse", "inhibitory",
                        {"weight":J_I, "delay":delay})

         nest.Connect(nodes_I, nodes,
                      {"rule": 'fixed_indegree', "indegree": C_I},
                      "inhibitory")
```

#### Adding stimulation and recording devices

- Same as before

```
In [46]: noise = nest.Create("poisson_generator", params={"rate": p_rate})

         # connect using all_to_all: one noise generator to all neurons
         nest.Connect(noise, nodes, syn_spec="excitatory")
```

```
spikes=nest.Create("spike_detector", 2,
                   params=[{"label": "Exc", "to_file": False},
                           {"label": "Inh", "to_file": False}])
spikes_E = spikes[:1]
spikes_I = spikes[1:]

N_rec   = 100   # Number of neurons to record from

# connect using all_to_all: all recorded excitatory neurons to one detector
nest.Connect(nodes_E[:N_rec], spikes_E)
nest.Connect(nodes_I[:N_rec], spikes_I)
```

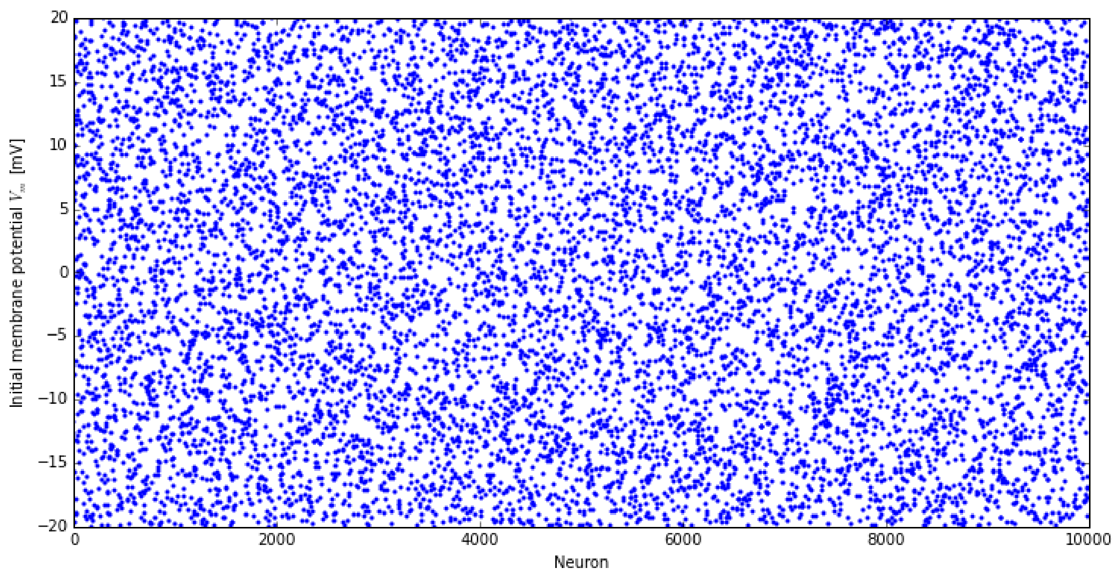
### 1.8.4 Inspecting the Network

**Initial membrane potentials**

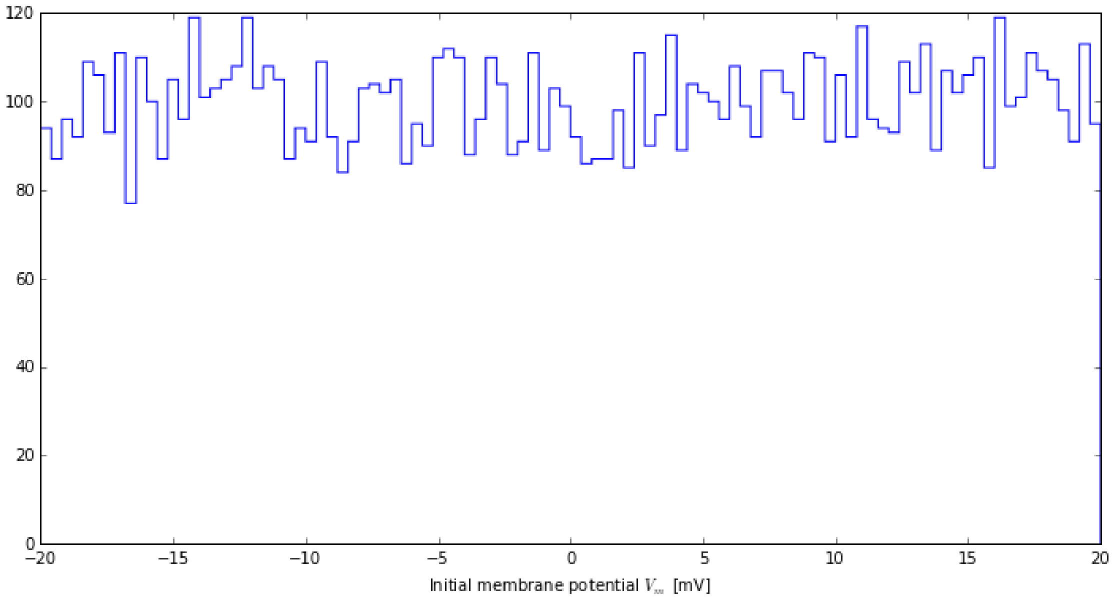
- Read out $V_m$ from all neurons
- Plot as scatter plot and histogram

```
In [47]: vm_ini = np.array(nest.GetStatus(nodes, 'V_m'))
```

```
In [48]: plt.plot(vm_ini, 'bo', markersize=3, markeredgecolor='none')
         plt.xlabel('Neuron')
         plt.ylabel('Initial membrane potential $V_m$ [mV]');
```




```
In [49]: plt.hist(vm_ini, bins=100, histtype='step')
         plt.xlabel('Initial membrane potential $V_m$ [mV]');
```




### Synaptic weights

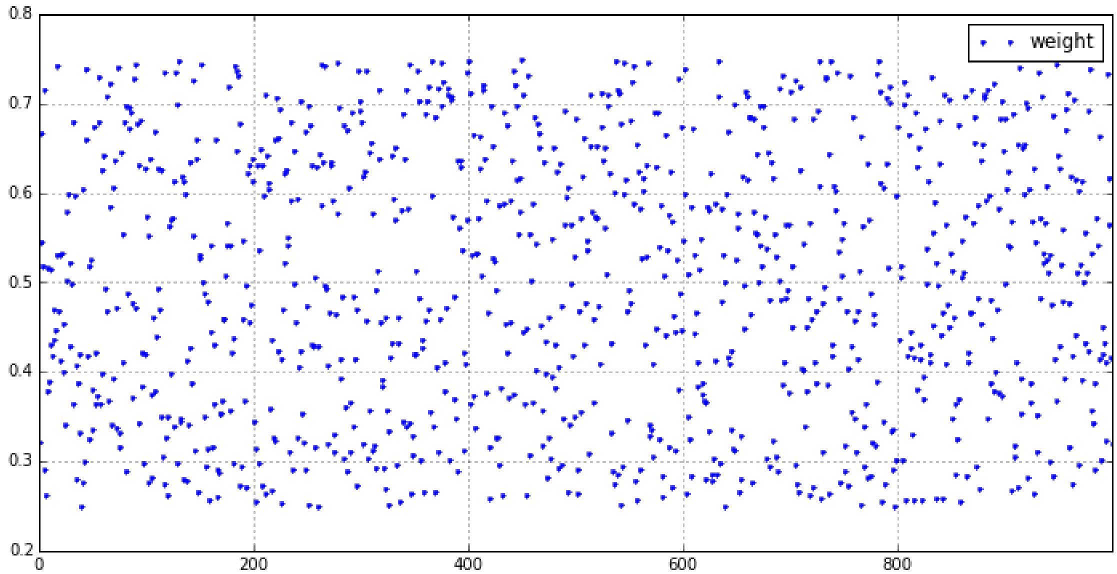
- We collect data on all outgoing synapses of type "excitatory"
- For the first 1000 neurons only
- We store as Pandas data frame

In [50]:
```
exc_conns = nest.GetConnections(source=nodes_E[:1000], synapse_model='excitatory')
exc_weights = pd.DataFrame({'weight': nest.GetStatus(exc_conns, 'weight')})

exc_weights.describe().T
```

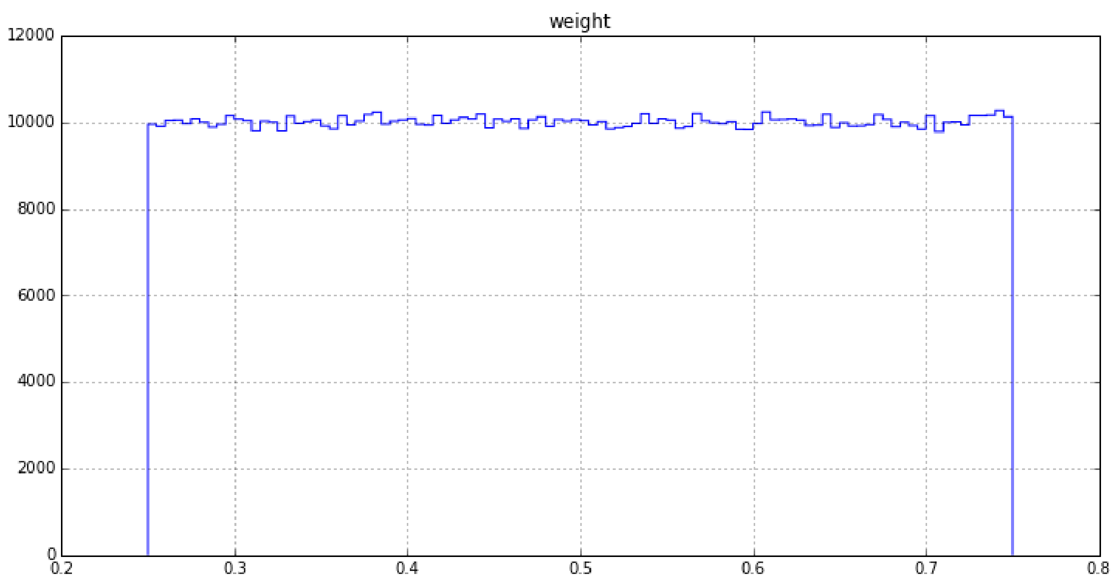
Out[50]:
```
            count      mean       std       min       25%      50%       75%  \
weight    1001230  0.500078  0.144371  0.250001  0.375266  0.49984  0.625029

          max
weight   0.75
```

- We scatter plot the first 1000
- We create a histogram of all

In [51]:
```
exc_weights[:1000].plot(style='.');
```




```
In [52]: exc_weights.hist(bins=100, histtype='step');
```




### 1.8.5 Simulating the network

```
In [53]: nest.Simulate(1000)
```

```
In [54]: sdE_data = pd.DataFrame(nest.GetStatus(spikes_E, 'events')[0])
         sdI_data = pd.DataFrame(nest.GetStatus(spikes_I, 'events')[0])

         e_spikes_plot = sdE_data[sdE_data.senders <= 40]
```

```
i_spikes_plot = sdI_data[(8000 < sdI_data.senders) & (sdI_data.senders <= 8010)]

plt.plot(e_spikes_plot.times, e_spikes_plot.senders, 'bo', markersize=5, markeredgecolor='none
plt.plot(i_spikes_plot.times, i_spikes_plot.senders-8000+40, 'ro', markersize=5, markeredgecol
plt.legend()
plt.xlabel('Time $t$ [ms]')
plt.yticks([])
plt.ylim(0, 51);
```