Scientific Python Tutorial – CCN Course 2013

How to code a neural network simulation

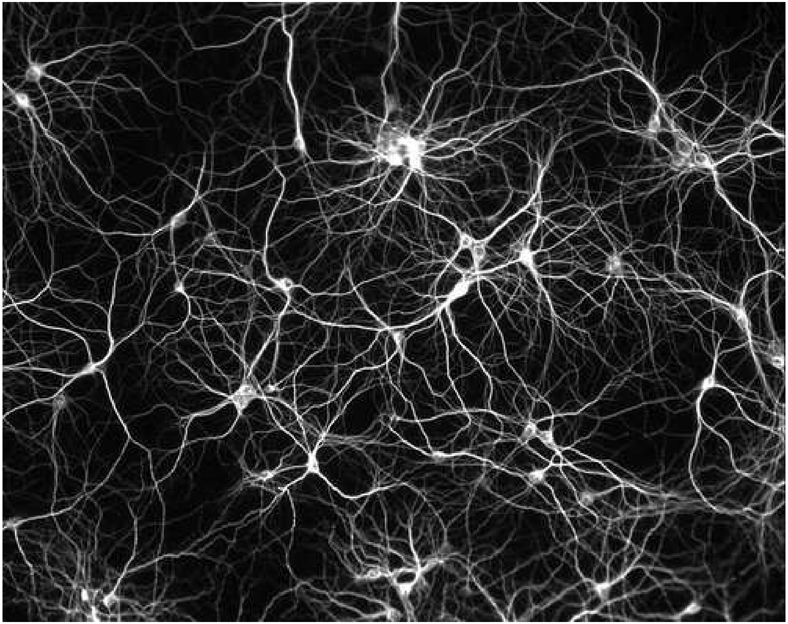
Malte J. Rasch

National Key Laboratory of Cognitive Neuroscience and Learning
Beijing Normal University
China

July 10, 2013

Goal of tutorial

- We will program a neural network simulation together.



We will practice on the way:

- Writing scripts

We will practice on the way:

- Writing scripts
- Usage of array notation

We will practice on the way:

- Writing scripts
- Usage of array notation
- How to integrate ODEs

We will practice on the way:

- Writing scripts
- Usage of array notation
- How to integrate ODEs
- How to plot results

We will practice on the way:

- Writing scripts
- Usage of array notation
- How to integrate ODEs
- How to plot results
- How to simulate neurons and synapses

We will practice on the way:

- Writing scripts
- Usage of array notation
- How to integrate ODEs
- How to plot results
- How to simulate neurons and synapses
- How to program a quite realistic network simulation

What has to be done in principle

- There are n **neurons**, excitatory and inhibitory, that are inter-connected with **synapses**.

What has to be done in principle

- There are n **neurons**, excitatory and inhibitory, that are inter-connected with **synapses**.
- The network gets some **input**

What has to be done in principle

- There are n **neurons**, excitatory and inhibitory, that are inter-connected with **synapses**.
- The network gets some **input**
- Each neuron and each synapse follows a particular **dynamics over time**.

What has to be done in principle

- There are n **neurons**, excitatory and inhibitory, that are inter-connected with **synapses**.
- The network gets some **input**
- Each neuron and each synapse follows a particular **dynamics over time**.
- The simulation solves the interplay of all components and e.g. yields **spiking activity** of the network for given inputs, which can be further analyzed (e.g. plotted)

We will proceed in 5 successive steps

1. Simulate a **single neuron** with current step input

We will proceed in 5 successive steps

1. Simulate a **single neuron** with current step input
2. Simulate a single neuron with **Poisson input**

We will proceed in 5 successive steps

1. Simulate a **single neuron** with current step input
2. Simulate a single neuron with **Poisson input**
3. Simulate **1000 neurons** (no recurrent connections)

We will proceed in 5 successive steps

1. Simulate a **single neuron** with current step input
2. Simulate a single neuron with **Poisson input**
3. Simulate **1000 neurons** (no recurrent connections)
4. Simulate a **recurrent network**

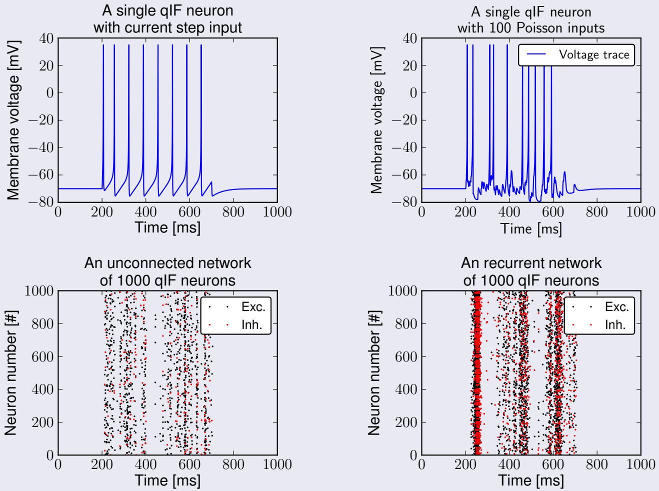
We will proceed in 5 successive steps

1. Simulate a **single neuron** with current step input
2. Simulate a single neuron with **Poisson input**
3. Simulate **1000 neurons** (no recurrent connections)
4. Simulate a **recurrent network**
5. Simulate a simple **orientation column**

We will proceed in 5 successive steps

1. Simulate a **single neuron** with current step input
2. Simulate a single neuron with **Poisson input**
3. Simulate **1000 neurons** (no recurrent connections)
4. Simulate a **recurrent network**
5. Simulate a simple **orientation column**

Result plots



Which neuron model to use?

Biophysical model (i.e. Hodgkin-Huxley model)

$$C_m \frac{dV_m}{dt} = -\frac{1}{R_m}(V_m - V_L) - \sum_i g_i(t)(V_m - E_i) + I$$

Including non-linear dynamics of many channels in $g_i(t)$

Which neuron model to use?

Biophysical model (i.e. Hodgkin-Huxley model)

$$C_m \frac{dV_m}{dt} = -\frac{1}{R_m}(V_m - V_L) - \sum_i g_i(t)(V_m - E_i) + I$$

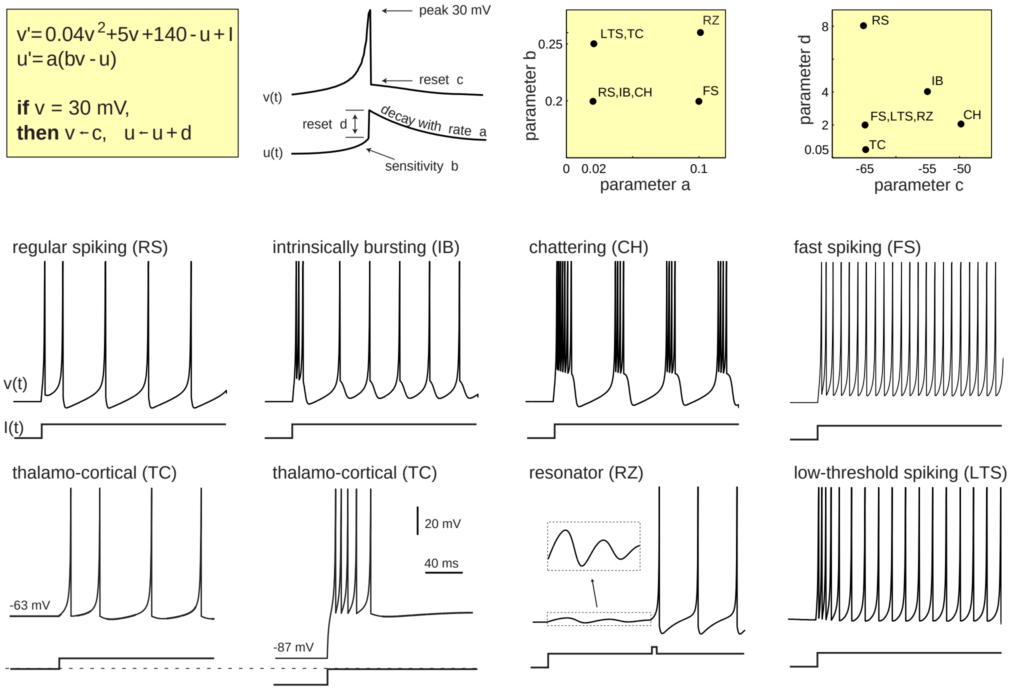
Including non-linear dynamics of many channels in $g_i(t)$

Mathematical simplification (Izhikevich, book chapter 8)

$$\begin{aligned}
\text{if} \quad v < 35 : & \\
& \dot{v} = (0.04v + 5)\, v + 150 - u - I \\
& \dot{u} = a\,(b\,v - u) \\
\text{if} \quad v \geq 35 : & \\
& v \leftarrow c \\
& u \leftarrow u + d
\end{aligned}$$

With $b = 0.2$, $c = -65$, and $d = 8$, $a = 0.02$ for excitatory neurons and $d = 2$, $a = 0.1$ for inhibitory neurons.

Neuron model



Step 1: Simulate a single neuron with injected current

Exercise 1

Simulate one excitatory neuron for 1000ms and plot the resulting voltage trace. Apply a current step ($I_{\text{app}} = 7\text{pA}$) between time 200ms and 700ms.

Step 1: Simulate a single neuron with injected current

Exercise 1

Simulate one excitatory neuron for 1000ms and plot the resulting voltage trace. Apply a current step ($I_{app} = 7$pA) between time 200ms and 700ms.

Neuron model:

if $\quad v < 35$:

$$\dot{v} = (0.04v + 5)\,v + 150 - u - I_{app}$$

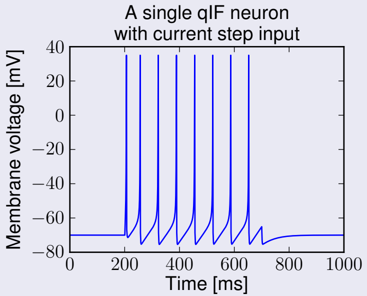
$$\dot{u} = a\,(b\,v - u)$$

if $\quad v \geq 35$:

$$v \leftarrow c$$

$$u \leftarrow u + d$$

with $d = 8,\; a = 0.02,\; b = 0.2,\; c = -65$



Step 1 in detail:

Open SPYDER and create a new file (script) that will simulate the neuron. Import the necessary modules (`from pylab import *`)

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5$ms, $a = 0.02$, $d = 8$, $\cdots$)

Step 1 in detail:

Open SPYDER and create a new file (script) that will simulate the neuron. Import the necessary modules (`from pylab import *`)

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5$ms, $a = 0.02$, $d = 8$, $\cdots$)
2. Reserve memory for voltage trace v and u (of length $T = 1000/\Delta t$) and set first element to -70 and -14, respectively.

Step 1 in detail:

Open SPYDER and create a new file (script) that will simulate the neuron. Import the necessary modules (`from pylab import *`)

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5$ms, $a = 0.02$, $d = 8$, $\cdots$)
2. Reserve memory for voltage trace v and u (of length $T = 1000/\Delta t$) and set first element to -70 and -14, respectively.
3. Loop over $T - 1$ time steps and do for each step t

Step 1 in detail:

Open SPYDER and create a new file (script) that will simulate the neuron. Import the necessary modules (`from pylab import *`)

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5$ms, $a = 0.02$, $d = 8$, $\cdots$)
2. Reserve memory for voltage trace v and u (of length $T = 1000/\Delta t$) and set first element to -70 and -14, respectively.
3. Loop over $T - 1$ time steps and do for each step t
    1. set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)

Step 1 in detail:

Open SPYDER and create a new file (script) that will simulate the neuron. Import the necessary modules (`from pylab import *`)

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5$ms, $a = 0.02$, $d = 8$, $\cdots$)
2. Reserve memory for voltage trace v and u (of length $T = 1000/\Delta t$) and set first element to -70 and -14, respectively.
3. Loop over $T - 1$ time steps and do for each step t
   1. set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)
   2. <u>if $v_t < 35$</u>: update element $t + 1$ of v and u according to

$$
\begin{aligned}
v_{t+1} &\leftarrow v_t + \Delta t \left\{(0.04\, v_t + 5)\, v_t - u_t + 140 + I_{\text{app}}\right\} \\
u_{t+1} &\leftarrow u_t + \Delta t\, a\, (b\, v_t - u_t)
\end{aligned}
$$

Step 1 in detail:

Open SPYDER and create a new file (script) that will simulate the neuron. Import the necessary modules (`from pylab import *`)

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5$ms, $a = 0.02$, $d = 8$, $\cdots$)
2. Reserve memory for voltage trace v and u (of length $T = 1000/\Delta t$) and set first element to -70 and -14, respectively.
3. Loop over $T - 1$ time steps and do for each step t
    1. set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)
    2. <u>if $v_t < 35$</u>: update element $t + 1$ of v and u according to

$$\begin{aligned} v_{t+1} &\leftarrow v_t + \Delta t \left\{ (0.04\, v_t + 5)\, v_t - u_t + 140 + I_{\text{app}} \right\} \\ u_{t+1} &\leftarrow u_t + \Delta t\, a\, (b\, v_t - u_t) \end{aligned}$$

    3. <u>if $v_t \geq 35$</u>: set the variables, $v_t \leftarrow 35$, $v_{t+1} \leftarrow c$, and $u_{t+1} \leftarrow u_t + d$.

Step 1 in detail:

Open SPYDER and create a new file (script) that will simulate the neuron. Import the necessary modules (`from pylab import *`)

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5$ms, $a = 0.02$, $d = 8, \cdots$)
2. Reserve memory for voltage trace v and u (of length $T = 1000/\Delta t$) and set first element to -70 and -14, respectively.
3. Loop over $T - 1$ time steps and do for each step t
   1. set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)
   2. <u>if $v_t < 35$</u>: update element $t + 1$ of v and u according to

$$\begin{aligned} v_{t+1} &\leftarrow v_t + \Delta t \left\{(0.04\, v_t + 5)\, v_t - u_t + 140 + I_{\text{app}}\right\} \\ u_{t+1} &\leftarrow u_t + \Delta t\, a\, (b\, v_t - u_t) \end{aligned}$$

   3. <u>if $v_t \geq 35$</u>: set the variables, $v_t \leftarrow 35$, $v_{t+1} \leftarrow c$, and $u_{t+1} \leftarrow u_t + d$.
4. Plot the voltage trace v versus t

Solution to step 1 (Python)

```python
from pylab import *

# 1) initialize parameters
tmax = 1000.
dt   = 0.5

# 1.1) Neuron / Network pars
a = 0.02 # RS,IB: 0.02, FS: 0.1
b = 0.2   # RS,IB,FS: 0.2
c = -65 # RS,FS: -65 IB: -55
d = 8. # RS: 8, IB: 4, FS: 2

# 1.2) Input pars
Iapp=10
tr=array([200.,700.])/dt # stm time

# 2) reserve memory
T = ceil(tmax/dt)
v = zeros(T)
u = zeros(T)
v[0] = -70 # resting potential
u[0] = -14 # steady state

# 3) for-loop over time
for t in arange(T-1):
    # 3.1) get input
    if t>tr[0] and t<tr[1]:
        I = Iapp
    else:
        I = 0


    if v[t]<35:
        # 3.2) update ODE
        dv = (0.04*v[t]+5)*v[t]+140-u[t]
        v[t+1] = v[t] + (dv+I)*dt
        du = a*(b*v[t]-u[t])
        u[t+1] = u[t] + dt*du
    else:
        # 3.3) spike !
        v[t] = 35
        v[t+1] = c
        u[t+1] = u[t]+d

# 4) plot voltage trace
figure()
tvec = arange(0.,tmax,dt)
plot(tvec,v,'b',label='Voltage trace')
xlabel('Time [ms]')
ylabel('Membrane voltage [mV]')
title("""A single qIF neuron
with current step input6""")
show()
```

Synapse model

Conductance based synaptic input

A simple synaptic input model would be

$$I_{\mathrm{syn}} = \sum_j w_j s_j (v - E_j)$$

where w_j is the weight of the jth synapse and E_j its reversal potential (for instance 0 mV for excitatory and -85 mV for inhibitory synapses).

Variable s_j implements the dynamics of the jth synapse:

$$\begin{aligned} \dot{s}_j &= -s_j/\tau_s \\ s_j &\leftarrow s_j + 1, \quad \text{if pre-synaptic neuron spikes} \end{aligned}$$

Synapse model

Conductance based synaptic input

A simple synaptic input model would be

$$I_{\text{syn}} = \sum_j w_j s_j (v - E_j)$$

where w_j is the weight of the jth synapse and E_j its reversal potential (for instance 0 mV for excitatory and -85 mV for inhibitory synapses).

Variable s_j implements the dynamics of the jth synapse:

$$\begin{aligned} \dot{s}_j &= -s_j/\tau_s \\ s_j &\leftarrow s_j + 1, \quad \text{if pre-synaptic neuron spikes} \end{aligned}$$

Optional: Synaptic depression

Change the update to

$$s_j \leftarrow s_j + h_j, \qquad h_j \leftarrow 1 - (1 + (U-1)h_j)e^{-\Delta t_j \tau_d},$$

with e.g. $U = 0.5$, $\tau_d = 500$ms. Δt_j is the interval between current and previous spike of neuron j.

Step 2: Single neuron with synaptic input

Exercise 2

Simulate the neuron model for 1000ms and plot the resulting voltage trace. Assume that 100 synapses are attached to the neuron, with each pre-synaptic neuron firing with a Poisson process of rate $f_{\text{rate}} = 2$ Hz between time 200ms and 700ms.

Step 2: Single neuron with synaptic input

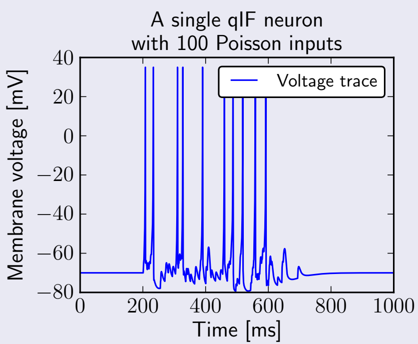
Exercise 2

Simulate the neuron model for 1000ms and plot the resulting voltage trace. Assume that 100 synapses are attached to the neuron, with each pre-synaptic neuron firing with a Poisson process of rate $f_{\text{rate}} = 2$ Hz between time 200ms and 700ms.

Synaptic input model:

$$
\begin{aligned}
I_{\text{syn}} &= \sum_j w_j^{\text{in}} s_j^{\text{in}}(t)(E_j^{\text{in}} - v(t)) \\
\dot{s}_j^{\text{in}} &= s_j^{\text{in}}/\tau_s \\
s_j^{\text{in}} &\leftarrow s_j^{\text{in}} + h_j, \quad \text{if synapse } j \text{ spikes}
\end{aligned}
$$

with $h_j = 1^*$, $\tau_s = 10$, $w_j^{\text{in}} = 0.07$, $E_j = 0$, $j = 1 \dots 100$. Poisson: Input synapse j spikes if $r_j(t) < f_{\text{rate}} \Delta t$, where $r_j(t) \in [0, 1]$ are uniform random numbers drawn for each step t.



Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

Proceed as follows:

1. Initialize new parameter values ($\tau_s = 10$, $f_{\text{rate}} = 0.002\text{ms}^{-1}$)

Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

Proceed as follows:

1. Initialize new parameter values ($\tau_s = 10$, $f_{\text{rate}} = 0.002\text{ms}^{-1}$)
2. Reserve memory and initialize the vectors $\mathbf{s}^{\text{in}} = (s_j^{\text{in}})$, $\mathbf{w}^{\text{in}} = (w_j^{\text{in}})$, and $\mathbf{E} = (E_j)$ with $n_{\text{in}} = 100$ constant elements (same values as in Step 1)

Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

Proceed as follows:

1. Initialize new parameter values ($\tau_s = 10$, $f_{\text{rate}} = 0.002\text{ms}^{-1}$)
2. Reserve memory and initialize the vectors $\mathbf{s}^{\text{in}} = (s_j^{\text{in}})$, $\mathbf{w}^{\text{in}} = (w_j^{\text{in}})$, and $\mathbf{E} = (E_j)$ with $n_{\text{in}} = 100$ constant elements (same values as in Step 1)
3. Inside the for-loop change/add the following:

Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

Proceed as follows:

1. Initialize new parameter values ($\tau_s = 10$, $f_{\text{rate}} = 0.002\text{ms}^{-1}$)
2. Reserve memory and initialize the vectors $\mathbf{s}^{\text{in}} = (s_j^{\text{in}})$, $\mathbf{w}^{\text{in}} = (w_j^{\text{in}})$, and $\mathbf{E} = (E_j)$ with $n_{\text{in}} = 100$ constant elements (same values as in Step 1)
3. Inside the for-loop change/add the following:
   1. Set $p_j = 1$ if $r_j \leq f_{\text{rate}}\Delta t$ (otherwise 0) during times of applied input. r_j is an uniform random number between 0 and 1. Use array notation to set the input for all n_{in} input synapses. Hint

Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

Proceed as follows:

1. Initialize new parameter values ($\tau_s = 10$, $f_{\text{rate}} = 0.002\text{ms}^{-1}$)
2. Reserve memory and initialize the vectors $\mathbf{s}^{\text{in}} = (s_j^{\text{in}})$, $\mathbf{w}^{\text{in}} = (w_j^{\text{in}})$, and $\mathbf{E} = (E_j)$ with $n_{\text{in}} = 100$ constant elements (same values as in Step 1)
3. Inside the for-loop change/add the following:
   1. Set $p_j = 1$ if $r_j \leq f_{\text{rate}} \Delta t$ (otherwise 0) during times of applied input. r_j is an uniform random number between 0 and 1. Use array notation to set the input for all n_{in} input synapses. Hint
   2. before the v_t update: Implement the conductance dynamics $\mathbf{s}$ and set I_{app} according to the input. Use array notation with dot "$\cdot$" product and element-wise "$\odot$" product. Hint

$$
\begin{aligned}
s_j^{\text{in}} &\leftarrow s_j^{\text{in}} + p_j \\
I_{\text{app}} &\leftarrow \mathbf{w}^{\text{in}} \cdot \left(\mathbf{s}^{\text{in}} \odot \mathbf{E}^{\text{in}}\right) - \left(\mathbf{w}^{\text{in}} \cdot \mathbf{s}^{\text{in}}\right) \odot v_t \\
s_j^{\text{in}} &\leftarrow \left(1 - \Delta t/\tau_s\right) s_j^{\text{in}}
\end{aligned}
$$

Solution to step 2 (Python)

```python
from pylab import *

# 1) initialize parameters
tmax = 1000.
dt   = 0.5

# 1.1) Neuron / Network pars
a = 0.02
b = 0.2
c = -65
d = 8.
tau_s = 10 # decay of synapses [ms]

# 1.2) Input pars
tr=array([200.,700])/dt
rate_in = 2 # input rate
n_in  = 100 # number of inputs
w_in  = 0.07 # input weights
W_in  = w_in*ones(n_in) # vector

# 2) reserve memory
T = ceil(tmax/dt)
v = zeros(T)
u = zeros(T)
v[0] = -70 # resting potential
u[0] = -14 # steady state
s_in  = zeros(n_in) # synaptic variable
E_in  = zeros(n_in) # rev potential
prate = dt*rate_in*1e-3 # abbrev

# 3) for-loop over time
for t in arange(T-1):
    # 3.1) get input
    if t>tr[0] and t<tr[1]:
        # NEW: get input Poisson spikes
        p = uniform(size=n_in)<prate;
    else:
        p = 0; # no input

    # NEW: calculate input current
    s_in  = (1 - dt/tau_s)*s_in + p
    I = dot(W_in, s_in*E_in)
    I -= dot(W_in, s_in)*v[t]

    if v[t]<35:
        # 3.2) update ODE
        dv = (0.04*v[t]+5)*v[t]+140-u[t]
        v[t+1] = v[t] + (dv+I)*dt
        du = a*(b*v[t]-u[t])
        u[t+1] = u[t] + dt*du
    else:
        # 3.3) spike !
        v[t] = 35
        v[t+1] = c
        u[t+1] = u[t]+d

# 4) plot voltage trace
figure()
tvec = arange(0.,tmax,dt)
plot(tvec,v,'b',label='Voltage trace')
xlabel('Time [ms]')
ylabel('Membrane voltage [mV]')
title("""A single qIF neuron
with %d Poisson inputs""" % n_in)
show()
```

Solution to step 2 with STP (Python)

```python
from pylab import *

# 1) initialize parameters
tmax = 1000.
dt   = 0.5

# 1.1) Neuron / Network pars
a = 0.02
b = 0.2
c = -65
d = 8.
tau_s = 10 # decay of synapses [ms]
tau_d = 500 # synaptic depression [ms]
stp_u = 0.5 # STP parameter

# 1.2) Input pars
tr=array([200.,700.])/dt
rate_in = 10 # input rate
n_in  = 1 # number of inputs
w_in  = 0.03 # input weights
W_in  = w_in*ones(n_in) # vector

# 2) reserve memory
T = ceil(tmax/dt)
v = zeros(T)
u = zeros(T)
v[0] = -70 # resting potential
u[0] = -14 # steady state
s_in  = zeros(n_in) # synaptic variable
E_in  = zeros(n_in) # rev potential
prate = dt*rate_in*1e-3 # abbrev
h = ones(n_in)
lastsp = -infty*ones(n_in)

# 3) for-loop over time
for t in arange(T-1):
    # 3.1) get input
    if t>tr[0] and t<tr[1]:
        # NEW: get input Poisson spikes
        p = uniform(size=n_in)<prate;

        #update synaptic depression
        tmp = exp(dt*(lastsp[p]-t)/tau_d)
        h[p] = 1 - (1+(stp_u -1)*h[p])*tmp
        lastsp[p] = t
    else:
        p = 0; # no input


    # NEW: calculate input current
    s_in = (1 - dt/tau_s)*s_in + p*h
    I = dot(W_in,s_in*E_in)
    I -= dot(W_in,s_in)*v[t]

    if v[t]<35:
        # 3.2) update ODE
        dv = (0.04*v[t]+5)*v[t]+140-u[t]
        v[t+1] = v[t] + (dv+I)*dt
        du = a*(b*v[t]-u[t])
        u[t+1] = u[t] + dt*du
    else:
        # 3.3) spike !
        v[t] = 35
        v[t+1] = c
        u[t+1] = u[t]+d

# 4) plot voltage trace
figure()
tvec = arange(0.,tmax,dt)
plot(tvec,v,'b',label='Voltage trace')
xlabel('Time [ms]')
ylabel('Membrane voltage [mV]')
title("""A single qIF neuron
with %d Poisson inputs""" % n_in)
show()
```

Step 3: Simulate 1000 neurons (not inter-connected)

Exercise 3

Simulate 1000 neurons for 1000 ms and plot the resulting spikes. Assume that each neuron receives (random) 10% of the 100 Poisson spike trains of rate $f_{\text{rate}} = 2$ Hz between time 200 ms and 700 ms. Note that the neurons are not yet inter-connected.

Step 3: Simulate 1000 neurons (not inter-connected)

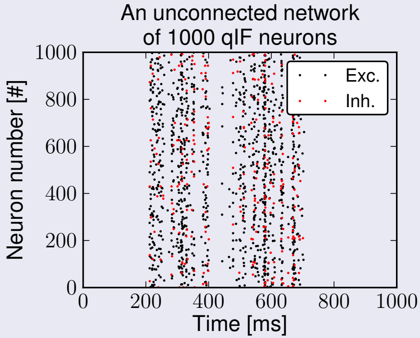
Exercise 3

Simulate 1000 neurons for 1000 ms and plot the resulting spikes. Assume that each neuron receives (random) 10% of the 100 Poisson spike trains of rate $f_{rate} = 2$ Hz between time 200 ms and 700 ms. Note that the neurons are not yet inter-connected.

Excitatory and inhibitory neurons:

A neuron is, with probability $p_{inh} = 0.2$, a (fast-spiking) inhibitory neuron ($a = 0.1$, $d = 2$), others are (regular spiking) excitatory neurons ($a = 0.02$ and $d = 8$).

Input weights of input synapse j to neuron i is set to $w^{in} = 0.07$ if connected (otherwise 0).



Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values ($n = 1000$)

Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values ($n = 1000$)
2. Initialize 2 logical vectors $\mathbf{k}_{\text{inh}}$ and $\mathbf{k}_{\text{exc}}$ of length n, where $\mathbf{k}_{\text{inh}}(i)$ is `True` with probability $p = 0.2$ (marking an inhibitory neuron) and `False` otherwise. And $\mathbf{k}_{\text{exc}} = \neg\mathbf{k}_{\text{inh}}$.

Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values ($n = 1000$)
2. Initialize 2 logical vectors $\mathbf{k}_{\text{inh}}$ and $\mathbf{k}_{\text{exc}}$ of length n, where $\mathbf{k}_{\text{inh}}(i)$ is `True` with probability $p = 0.2$ (marking an inhibitory neuron) and `False` otherwise. And $\mathbf{k}_{\text{exc}} = \neg\mathbf{k}_{\text{inh}}$.
3. Reserve memory and initialize $v_{i,t}$, $u_{i,t}$ (now being $T \times n$ matrices). Set parameter vectors $\mathbf{a}$ and $\mathbf{d}$ according to $\mathbf{k}_{\text{exc}}$ and $\mathbf{k}_{\text{inh}}$.

Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values ($n = 1000$)
2. Initialize 2 logical vectors $\mathbf{k}_{\text{inh}}$ and $\mathbf{k}_{\text{exc}}$ of length n, where $\mathbf{k}_{\text{inh}}(i)$ is `True` with probability $p = 0.2$ (marking an inhibitory neuron) and `False` otherwise. And $\mathbf{k}_{\text{exc}} = \neg\mathbf{k}_{\text{inh}}$.
3. Reserve memory and initialize $v_{i,t}$, $u_{i,t}$ (now being $T \times n$ matrices). Set parameter vectors $\mathbf{a}$ and $\mathbf{d}$ according to $\mathbf{k}_{\text{exc}}$ and $\mathbf{k}_{\text{inh}}$.
4. The weights $w_{ij}^{\text{in}} = 0.07$ now form a $n \times n_{\text{in}}$ matrix. Set 90 % random elements to 0 to account for the connection probability.

Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values ($n = 1000$)
2. Initialize 2 logical vectors $\mathbf{k}_{\text{inh}}$ and $\mathbf{k}_{\text{exc}}$ of length n, where $\mathbf{k}_{\text{inh}}(i)$ is `True` with probability $p = 0.2$ (marking an inhibitory neuron) and `False` otherwise. And $\mathbf{k}_{\text{exc}} = \neg \mathbf{k}_{\text{inh}}$.
3. Reserve memory and initialize $v_{i,t}$, $u_{i,t}$ (now being $T \times n$ matrices). Set parameter vectors $\mathbf{a}$ and $\mathbf{d}$ according to $\mathbf{k}_{\text{exc}}$ and $\mathbf{k}_{\text{inh}}$.
4. The weights $w_{ij}^{\text{in}} = 0.07$ now form a $n \times n_{\text{in}}$ matrix. Set 90 % random elements to 0 to account for the connection probability.
5. Inside the for-loop change/add the following:
    1. Same update equations (for $v_{i,t+1}$ and $u_{i,t+1}$) but use array notation.

Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values ($n = 1000$)
2. Initialize 2 logical vectors $\mathbf{k}_{\text{inh}}$ and $\mathbf{k}_{\text{exc}}$ of length n, where $\mathbf{k}_{\text{inh}}(i)$ is `True` with probability $p = 0.2$ (marking an inhibitory neuron) and `False` otherwise. And $\mathbf{k}_{\text{exc}} = \neg \mathbf{k}_{\text{inh}}$.
3. Reserve memory and initialize $v_{i,t}$, $u_{i,t}$ (now being $T \times n$ matrices). Set parameter vectors **a** and **d** according to $\mathbf{k}_{\text{exc}}$ and $\mathbf{k}_{\text{inh}}$.
4. The weights $w_{ij}^{\text{in}} = 0.07$ now form a $n \times n_{\text{in}}$ matrix. Set 90 % random elements to 0 to account for the connection probability.
5. Inside the for-loop change/add the following:
    1. Same update equations (for $v_{i,t+1}$ and $u_{i,t+1}$) but use array notation.
6. Plot the spike raster. Plot black dots at $\{(t,i) | v_{it} \geq 35\}$ for excitatory neurons i. Use red dots for inhibitory neurons.

Solution to step 3 (Python)

```python
from pylab import *

# 1) initialize parameters
tmax = 1000.
dt   = 0.5

# 1.1) Neuron / Network pars
n = 1000 # number of neurons
pinh = 0.2 # prob of inh neuron
inh = (uniform(size=n)<pinh) # whether inh.
exc = logical_not(inh)
a = inh.choose(0.02,0.1)# exc=0.02,inh=0.1
b = 0.2
c = -65
d = inh.choose(8,2) # exc=8,inh=2
tau_s = 10

# 1.2) Input pars
tr=array([200.,700])/dt
rate_in = 2
n_in   = 100
w_in   = 0.07
pconn_in = 0.1 # input conn prob
C = uniform(size=(n,n_in))<pconn_in
W_in = C.choose(0,w_in) # matrix

# 2) reserve memory
T = ceil(tmax/dt)
v = zeros((T,n)) # now matrix
u = zeros((T,n)) # now matrix
v[0] = -70 # set 1st row
u[0] = -14
s_in   = zeros(n_in)
E_in   = zeros(n_in)
prate = dt*rate_in*1e-3

# 3) for-loop over time
for t in arange(T-1):
    # 3.1) get input
    if t>tr[0] and t<tr[1]:
        p = uniform(size=n_in)<prate;
    else:
        p = 0;

    s_in = (1 - dt/tau_s)*s_in + p
    I = W_in.dot(s_in*E_in)
    I -= W_in.dot(s_in)*v[t]

    # NEW: handle all neurons
    fired = v[t]>=35

    # 3.2) update ODE, simply update all
    dv = (0.04*v[t]+5)*v[t]+140-u[t]
    v[t+1] = v[t] + (dv+I)*dt
    du = a*(b*v[t]-u[t])
    u[t+1] = u[t] + dt*du

    # 3.3) spikes !
    v[t][fired] = 35
    v[t+1][fired] = c
    u[t+1][fired] = u[t][fired]+d[fired]

# 4) plotting
# NEW: get spikes and plot
tspk,nspk = nonzero(v==35)
idx_i = in1d(nspk,nonzero(inh)[0]) # find inh
idx_e = logical_not(idx_i) # all others are exc

figure()
plot(tspk[idx_e]*dt,nspk[idx_e],'k.',
     label='Exc.',markersize=2)
plot(tspk[idx_i]*dt,nspk[idx_i],'r.',
     label='Inh.',markersize=2)
xlabel('Time [ms]')
ylabel('Neuron number [\#]')
xlim((0,tmax))
title("""An unconnected network
of %d qIF neurons""" % n)
legend(loc='upper right')
show()
```

Step 4: Simulate recurrent network

Exercise 4

Simulate 1000 neurons as before but with added recurrent connections.

Step 4: Simulate recurrent network

Exercise 4

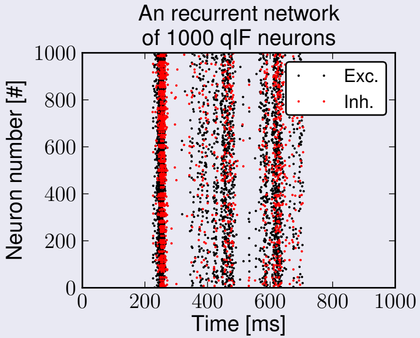
Simulate 1000 neurons as before but with added recurrent connections.

Recurrent synaptic activations

A neuron i is sparsely connected to a neuron j (with probability $p_{\text{conn}} = 0.1$). Thus neuron i receives an additional current I_i^{syn} of the form:

$$I_i^{\text{syn}} = \sum_{j=1}^{n} w_{ij} s_j(t) \left(E_j - v_i(t)\right)$$

Weights are Gamma distributed ($w_{\text{avg}} = 0.005$ and $g_{\text{sc}} = 0.002$). Set the inhibitory to excitatory connections twice as strong on average.



Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables ($\mathbf{s} = (s_j)$, E_j). Set $E_j = -85$ if j is an inhibitory neuron (otherwise 0).

Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables ($\mathbf{s} = (s_j)$, E_j). Set $E_j = -85$ if j is an inhibitory neuron (otherwise 0).

2. Reserve memory and initialize weights $W = (w_{ij})$ to zero. Randomly choose 10% of the matrix elements.

Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables ($\mathbf{s} = (s_j)$, E_j). Set $E_j = -85$ if j is an inhibitory neuron (otherwise 0).
2. Reserve memory and initialize weights $W = (w_{ij})$ to zero. Randomly choose 10% of the matrix elements.
3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale $g_{\mathrm{sc}} = 0.002$ and shape $g_{\mathrm{sh}} = \frac{w_{\mathrm{avg}}}{g_{sc}}$, with $w_{\mathrm{avg}} = 0.005$. Hint

Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables ($\mathbf{s} = (s_j)$, E_j). Set $E_j = -85$ if j is an inhibitory neuron (otherwise 0).
2. Reserve memory and initialize weights $W = (w_{ij})$ to zero. Randomly choose 10% of the matrix elements.
3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale $g_{\mathrm{sc}} = 0.002$ and shape $g_{\mathrm{sh}} = \frac{w_{\mathrm{avg}}}{g_{sc}}$, with $w_{\mathrm{avg}} = 0.005$. Hint
4. Make the weight matrix "sparse" to speed up computations. Hint

Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables ($\mathbf{s} = (s_j)$, E_j). Set $E_j = -85$ if j is an inhibitory neuron (otherwise 0).
2. Reserve memory and initialize weights $W = (w_{ij})$ to zero. Randomly choose 10% of the matrix elements.
3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale $g_{\text{sc}} = 0.002$ and shape $g_{\text{sh}} = \frac{w_{\text{avg}}}{g_{sc}}$, with $w_{\text{avg}} = 0.005$. Hint
4. Make the weight matrix "sparse" to speed up computations. Hint
5. Scale weights from inh. to exc. neurons by the factor of 2. Hint

Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables ($\mathbf{s} = (s_j)$, E_j). Set $E_j = -85$ if j is an inhibitory neuron (otherwise 0).
2. Reserve memory and initialize weights $W = (w_{ij})$ to zero. Randomly choose 10% of the matrix elements.
3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale $g_{sc} = 0.002$ and shape $g_{sh} = \frac{w_{avg}}{g_{sc}}$, with $w_{avg} = 0.005$. Hint
4. Make the weight matrix "sparse" to speed up computations. Hint
5. Scale weights from inh. to exc. neurons by the factor of 2. Hint
6. Inside the for-loop change/add the following:
   1. add the equations for recurrent synaptic dynamics s_j and add I_{syn} to the total applied current.

$$
\begin{aligned}
s_j &\leftarrow s_j + 1, \quad \text{if } v_j(t-1) \geq 35 \\
\mathbf{I}^{syn} &\leftarrow W \cdot (\mathbf{s} \odot \mathbf{E}) - (W \cdot \mathbf{s}) \odot \mathbf{v} \\
s_j &\leftarrow (1 - \Delta t/\tau_s)\, s_j
\end{aligned}
$$

Solution to step 4

```python
from pylab import *
from scipy.sparse import csr_matrix

# 1) initialize parameters
tmax = 1000.
dt   = 0.5

# 1.1) Neuron / Network pars
n = 1000
pinh = 0.2
inh = (uniform(size=n)<pinh)
exc = logical_not(inh)
a = inh.choose(0.02,0.1)
b = 0.2
c = -65
d = inh.choose(8,2)
tau_s = 10

# NEW recurrent parameter
w = 0.005   # average recurrent weight
pconn = 0.1   # recurrent connection prob
scaleEI = 2 # scale I->E
g_sc = 0.002 # scale of gamma
E = inh.choose(0,-85)
# NEW: make weight matrix
W = zeros((n,n))
C = uniform(size=(n,n))
idx = nonzero(C<pconn) # sparse connectivity
W[idx] = gamma(w/g_sc, scale=g_sc, size=idx[0].size)
W[ix_(exc,inh)] *= scaleEI #submat indexing
W = csr_matrix(W)   # make row sparse

# 1.2) Input pars
tr=array([200.,700.])/dt
rate_in = 2
n_in   = 100
w_in   = 0.07
pconn_in = 0.1
C = uniform(size=(n,n_in))<pconn_in
W_in = C.choose(0,w_in)

# 2) reserve memory
T = ceil(tmax/dt)
v = zeros((T,n))
u = zeros((T,n))
v[0] = -70
u[0] = -14
s_in  = zeros(n_in)
E_in  = zeros(n_in)
prate = dt*rate_in*1e-3
s = zeros(n) # rec synapses

# 3) for-loop over time
for t in arange(T-1):
    # 3.1) get input
    if t>tr[0] and t<tr[1]:
        p = uniform(size=n_in)<prate;
    else:
        p = 0;

    s_in = (1 - dt/tau_s)*s_in + p
    I = W_in.dot(s_in*E_in)
    I -= W_in.dot(s_in)*v[t]

    fired = v[t]>=35

    # NEW: recurrent input
    s =  (1 - dt/tau_s)*s + fired
    Isyn = W.dot(s*E) - W.dot(s)*v[t]
    I += Isyn # add to input vector

    # 3.2) update ODE
    dv = (0.04*v[t]+5)*v[t]+140-u[t]
    v[t+1] = v[t] + (dv+I)*dt
    du = a*(b*v[t]-u[t])
    u[t+1] = u[t] + dt*du

    # 3.3) spikes !
    v[t][fired] = 35
    v[t+1][fired] = c
    u[t+1][fired] = u[t][fired]+d[fired]

# 4) plotting
tspk,nspk = nonzero(v==35)
```

Step 5: Simulate an orientation column

Exercise 5

Restructure the connection matrix and the input to simulate an orientation column. That is all E-E neurons only connect to neighboring neurons and the network resembles a 1D ring.

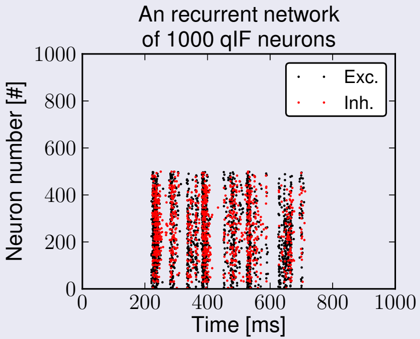
Step 5: Simulate an orientation column

Exercise 5

Restructure the connection matrix and the input to simulate an orientation column. That is all E-E neurons only connect to neighboring neurons and the network resembles a 1D ring.

Ring structure

A neuron i is still sparsely connected to a neuron j but now with probability $p_{\text{conn}} = 0.4$. However, if all neurons are arranged on a ring from 0 to 2π exc-to-exc connections are only possible if two neurons are nearer than $\pi/4$. Input is only delivered to a half of neurons (e.g. from 0 to *pi*). Use the same input connection probability as before.



Step 5 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Set indexes of the weight matrix to zero, which belong to exc-exc connections further apart that $\pi/4$. Hint: One can use `scipy.linalg.circulant`

Step 5 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Set indexes of the weight matrix to zero, which belong to exc-exc connections further apart that $\pi/4$. Hint: One can use `scipy.linalg.circulant`
2. Change the input so that only half (e.g. from 0 to *pi*) of the neurons receive input (again with probability 0.2). neurons

Solution to step 5

```python
from pylab import *
from scipy.sparse import csr_matrix
from scipy.linalg import circulant
# 1) initialize parameters
tmax = 1000.
dt   = 0.5

# 1.1) Neuron / Network pars
n = 1000
pinh = 0.2
inh = (uniform(size=n)<pinh)
exc = logical_not(inh)
a = inh.choose(0.02,0.1)
b = 0.2
c = -65
d = inh.choose(8,2)
tau_s = 10

width = pi/4 # half-width of the orientation tuning
w = 0.005
pconn = 0.4  # set a bit higher
scaleEI = 2
g_sc = 0.002
E = inh.choose(0,-85)
W = zeros((n,n))
C = uniform(size=(n,n))
idx = nonzero(C<pconn)
W[idx] = gamma(w/g_sc,scale=g_sc,size=idx[0].size)
W[ix_(exc,inh)] *= scaleEI
theta = linspace(0,2*pi,n) # NEW
R = circulant(cos(theta))>cos(width) #NEW
W[:,exc] = where(R[:,exc],W[:,exc],0) # NEW
W = csr_matrix(W)

# 1.2) Input pars
tr=array([200.,700])/dt
rate_in = 2
inwidth = pi/2
w_in  = 0.07
pconn_in = 0.2
n_in = 100
C = uniform(size=(n,n_in))<pconn_in
W_in = C.choose(0,w_in)
W_in[n/2:,:] = 0 # NEW

# 2) reserve memory
T = ceil(tmax/dt)
v = zeros((T,n))
u = zeros((T,n))
v[0] = -70
u[0] = -14
s_in  = zeros(n_in)
E_in  = zeros(n_in)
prate = dt*rate_in*1e-3
s = zeros(n) # rec synapses

# 3) for-loop over time
for t in arange(T-1):
    # 3.1) get input
    if t>tr[0] and t<tr[1]:
        p = uniform(size=n_in)<prate;
    else:
        p = 0;

    s_in = (1 - dt/tau_s)*s_in + p
    I = W_in.dot(s_in*E_in)
    I -= W_in.dot(s_in)*v[t]

    fired = v[t]>=35

    # NEW: recurrent input
    s =  (1 - dt/tau_s)*s + fired
    Isyn = W.dot(s*E) - W.dot(s)*v[t]
    I += Isyn # add to input vector

    # 3.2) update ODE
    dv = (0.04*v[t]+5)*v[t]+140-u[t]
    v[t+1] = v[t] + (dv+I)*dt
    du = a*(b*v[t]-u[t])
    u[t+1] = u[t] + dt*du

    # 3.3) spikes !
    v[t][fired] = 35
    v[t+1][fired] = c
```

Congratulation !

You have just coded and simulated a quite realistic network model !

Hint for generating random number

Use the `uniform` function to generate arrays of random numbers between 0 and 1.

```
from numpy.random import uniform
n_in = 100
r = uniform(size=n_in)
```

To set indexes i of a vector v to a with a probability p and otherwise to b one can use the method `choose`

```
r = uniform(size=n_in)
v = (r<p).choose(b,a)
```

back

Hint for generating gamma distributed random number

Use the `numpy.random.gamma` function to generate arrays of random numbers which are gamma distributed.

```
from numpy.random import gamma

g_shape, g_scale, n = 0.003, 2, 1000
r = gamma(g_shape,g_scale,n)
```

back

Hint for making sparse matrices

There are several forms of sparse matrices in the module `scipy.sparse`. The one which is interesting for our purposes is the "row-wise" sparse matrix (see the documentation of `scipy.sparse` for more information).

```
from pylab import *
from scipy.sparse import csr_matrix

R = uniform(size=(100,100)) # example matrix
W = where(R<0.1,R,0)        # mostly 0
W2 = csr_matrix(W)          # make sparse matrix

v = uniform(size=100)       # example vector
x = W2.dot(v)               # matrix dot product
```

back

Hint for submatrix indexing

`numpy.array` provides a shortcut for (MATLAB-like) submatrix indexing. Assume one has a matrix W and wanted to add 1 to add 1 to a selection of rows and columns. One could use the convenient function `ix_` and write

```
from pylab import *

W = uniform(size=(10,10))         # example matrix
i_row = uniform(size=10)<0.5      # select some rows
i_col = uniform(size=10)<0.5      # select some cols

W(ix_(i_row,i_col)) += 1          # add 1 to the elements
```

back

Hint for using dot product

Use the `dot` method of an `numpy.array` to compute the dot product.
Caution: The operator * yields an *element-wise* multiplication!

```
from pylab import *
a = array([1.,2,3,4])
b = array([1.,1,5,5])
c = a*b       # element-wise !
c.size
4
d = a.dot(b) # scalar product
d.size
1
```

back