NEURON - tutorial C (part 3) and tutorial E of Gillies & Sterratt

http://www.anc.ed.ac.uk/school/neuron/

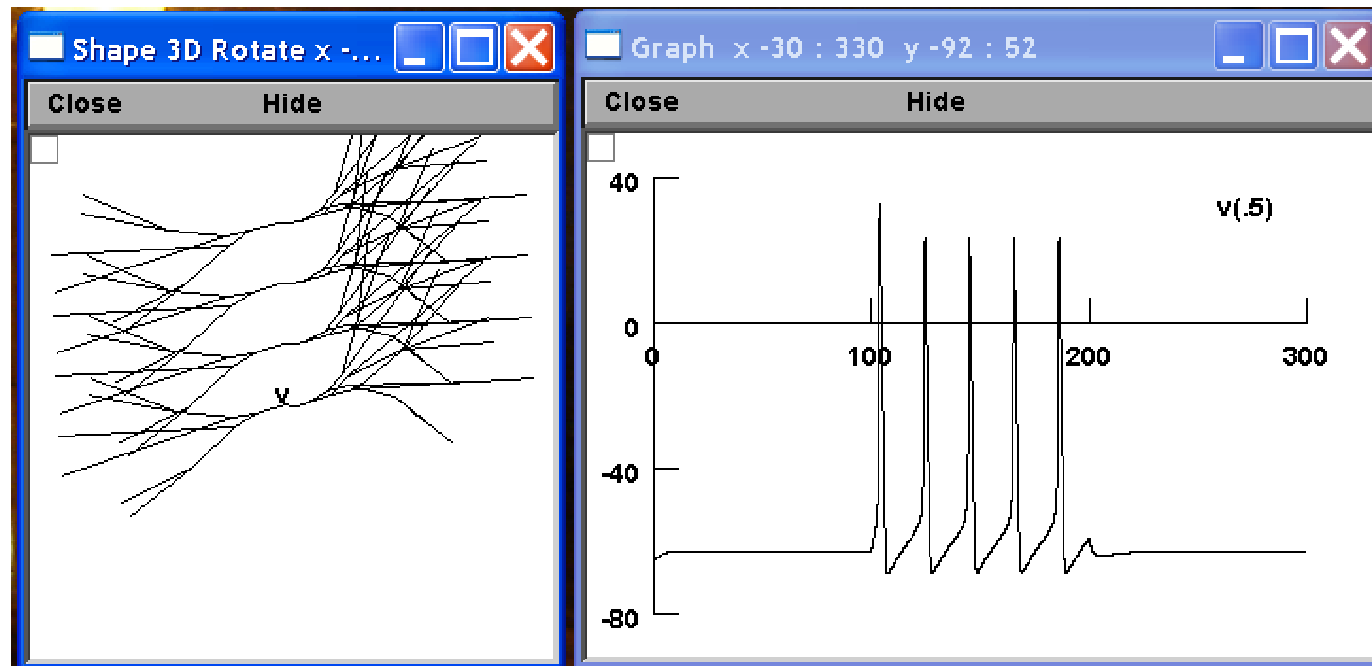
COSC422 – lecture 9

How to connect neurons using **`NetCon`**

Methods of getting data out of neuron to be stored, visualised or analysed in other packages

What we've got so far: **`sthC2.hoc`**

- The final four neurons, each with a full dendritic tree morphology are shown here in a shape plot (image on the left).
- Next to it is the voltage trace in one of the model neurons, i.e. **`SThcells[0].soma`**, as a result of the current pulse injection (recall the cells are not connected yet).



Connecting neurons with a delay line

- Unless we need for some reason simulate explicitly propagation of spikes along axons, then we can connect our neurons together without considering axons at all !

- That's what we are going to do now. We would save a lot of computation time doing it this way.

- Since the information content of signals passing from one neuron to another is carried by the times of spikes arriving at synapses, we will model the connection from one neuron to another as a **delay line**.

The delay line

- The voltage in the soma of the presynaptic cell *j* is continuously monitored.
- If the voltage goes over a defined threshold, this signals the occurrence of an output spike (action potential).
- The delay line then signals this occurrence to the synaptic contact on the postsynaptic neuron *i* at a defined time later (i.e. after some delay Δt).
- The delay is calculated based on the known velocity of spike propagation along axons and the distance between the two neurons as $\Delta t = \text{distance} / \text{velocity}$.

Event based simulation model

- We have not yet connected our neurons together. Currently, they are operating as four independent subthalamic neurons, each with an electrode injecting a current pulse into the soma.

- The modern form of connecting neurons together is in an *event based simulation model* (events are usually things like spikes).

- By using an event based model, we can dramatically reduce the amount of computation. This is important in simulation time optimisation and in simulations on parallel machines as only spike times need to be sent between processors.

Connecting neurons together – NetCon

- First we must add an additional public object variable to our neuron template, the **`nclist`**, to be accessible from outside the template.
- We have to declare a new object variable **`nclist`** that will refer to a list that will hold an arbitrary number of NetCon objects.
- So, now we begin our subthalamic neuron template with:

```
begintemplate SThCell
public soma, treeA, treeB, nclist
create soma, treeA[1], treeB[1]
objectvar f, nclist
```

Connecting neurons together – **`List`**

- Then we continue with the init() procedure like this:

```
proc init() {
        local i,me,child1,child2
        create soma
        nclist = new List()
```

- The only thing we have to change is to add an object variable **`nclist`**, make it public, and in the **`init()`** procedure, associate it with a **`new List`**.
- In NEURON, a **`List`** is an object that holds a list of other objects. The advantage of a list is that we don't have to specify in advance how big it will grow, as we have to for an array.

Changing the number of electrodes

- At present, we insert current clamps into all of four somas as follows:

```
objectvar stim[nSThcells]

for i = 0, nSThcells-1 SThcells[i].soma {
     stim[i] = new IClamp(0.5)
     stim[i].del = 100
     stim[i].dur = 100
     stim[i].amp = 0.1
}
```

Changing the number of electrodes

- Let from 100 to 200 ms, only the neuron **`SThcells[1]`** will be generating spikes.

- The modified code looks like this:

```
objectvar stim[nSThcells]
i = 1
SThcells[i].soma {
stim[i] = new IClamp(0.5)
stim[i].del = 100
stim[i].dur = 100
stim[i].amp = 0.1
}
```

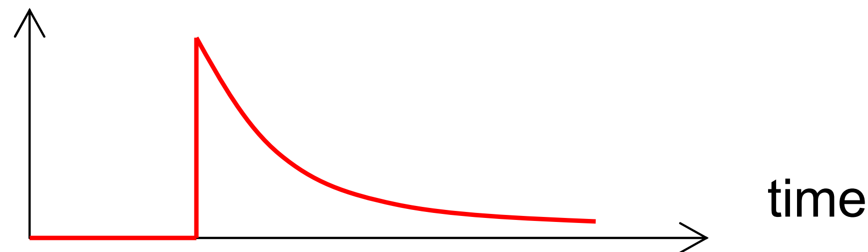
Creating synapses

- First, we only want to connect the stimulated neuron **`SThcells[1]`** to model neuron **`SThcells[0]`** and observe the Excitatory Post Synaptic Potentials at the soma of neuron 0.
- In order to connect neurons, we must create one or more synapses.
- Like IClamp stimulation, a synapse is an object that can be positioned anywhere on a neuron.
- For example, at the end of our current code we can define a new array of objects for our synapses. Let's have a maximum of 10 synapses:

```
maxsyn = 10
objectvar syn[maxsyn]
```

`ExpSyn` ("exponential synapse")

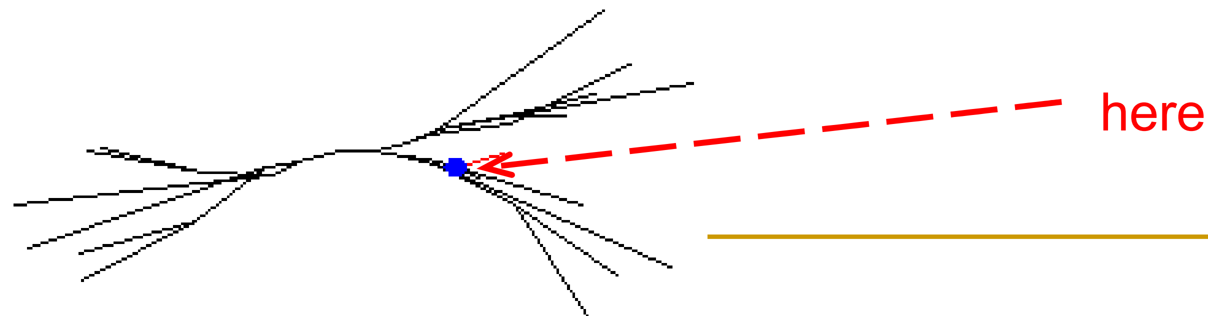
- The built in synaptic type **`ExpSyn`** is a synapse whose conductance instantaneously rises on receiving a spike and then exponentially decays.



- We position and create the synapse at a chosen section of a neuron:

```
SThcells[0].treeA[7] syn[0] = new ExpSyn(0)
```

- Thus, a synapse 0 will be placed at the branch 7 of the neuron 0:



Defining the source of events for synapses

- To create a new **NetCon** object (the source for a synapse), we use the command format:

```
new NetCon(&source_v,synapse,threshold,delay,weight)
```

- ***source_v*** is the source voltage (from **SThcells[1].soma**);
- ***synapse*** is the object variable that refers to the synaptic object receiving the events (in our case ***syn[0]***);
- ***threshold*** is the threshold, which the voltage must reach for it to be considered that a spike has occurred;
- ***delay*** is the connection delay, and
- ***weight*** is the connection weight strength of the synapse.

Connecting synapse

- So, to connect **SThcells[1]** to the dendritic branch 7 of **treeA** on subthalamic neuron **SThcells[0]** we add the command:

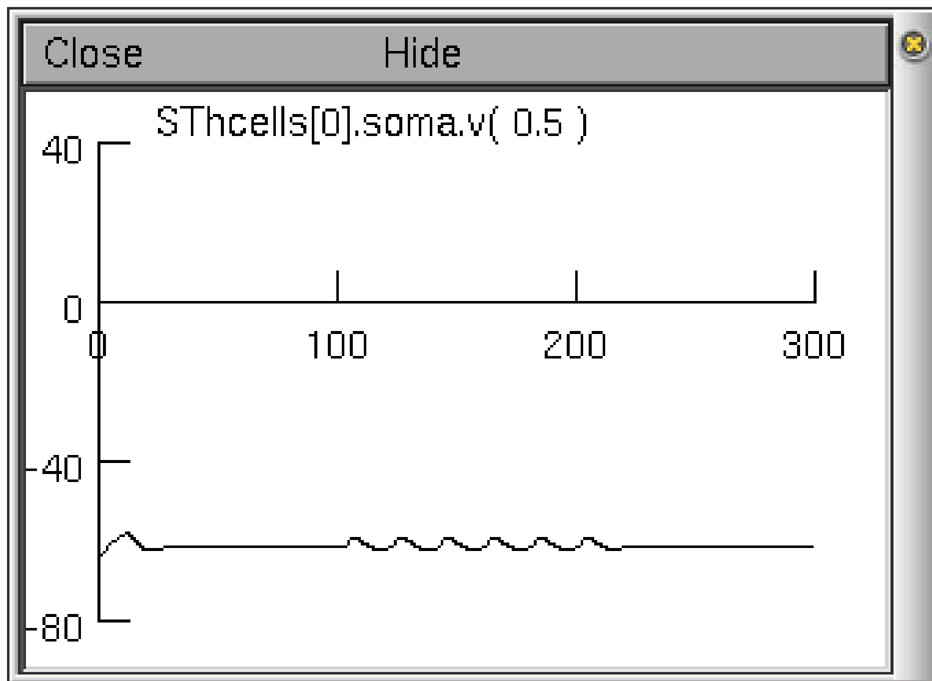
```
SThcells[1].soma SThcells[0].nclist.append(new
NetCon(&v(1), syn[0], -20, 1, 0.5))
```

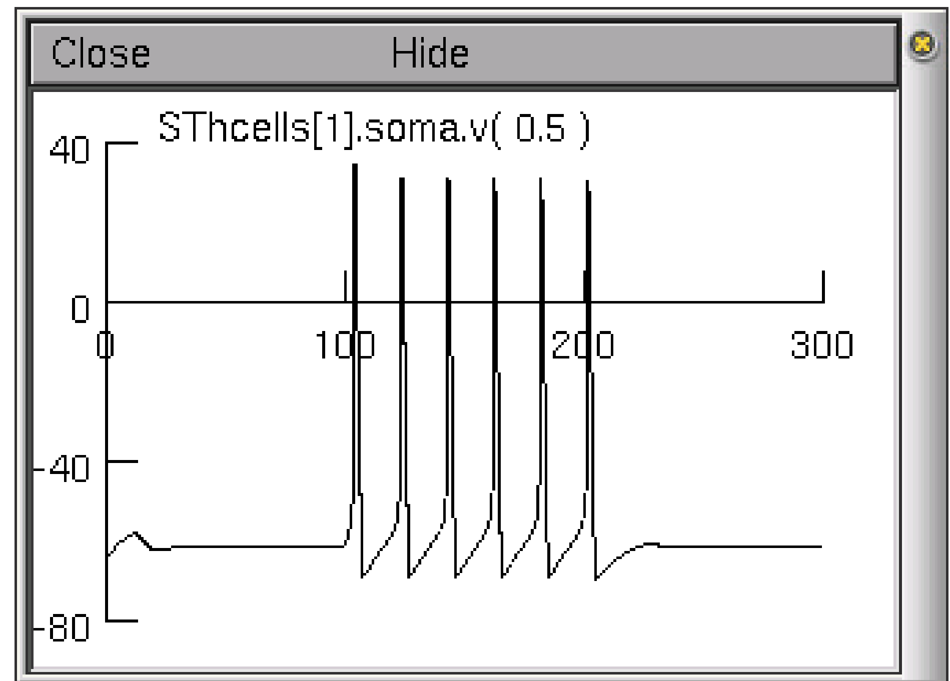
- First this command accesses **SThcells[1].soma**
- then the **nclist** of **SThcells[0]** has a new NetCon object appended. This NetCon object has a source voltage of **SThcells[1].soma.v(1)** .
- The NetCon object applies to **syn[0]** which we have already attached to **SThcells[0].treeA[7]** .
- Our threshold for action potentials is –20mV, our delay 1ms, and our synaptic weight 0.5.

Simulation of `sthC3.hoc`

- The last line of the code is: **`access SThcells[0].soma`**

- If we run the simulation and plot the voltage at **`SThcells[0].soma`** we see EPSPs resulting from the spikes of neuron **`SThcells[1]`**:





Dealing with lists

- The command **`nclist.append(obj)`** appends the object specified by the object variable **`obj`** to the list **`nclist`**. E.g.

```
SThcells[1].soma SThcells[0].nclist.append(new
NetCon(&v(1), syn[0], -20, 1, 0.5))
```

- There are number of commands to manipulate nclists. One of them is

```
nclist.count()
```

- which returns the number of items in the list **`nclist`**.

Dealing with lists

- The command **`nclist.object(i)`**
- returns the object at index **`i`** in the list **`nclist`**.
- We can put these commands together to change properties of the synaptic connections. For example:

```
for i = 0, SThcells[0].nclist.count()-1 {
SThcells[0].nclist.object(i).weight = 0.6 }
```

- Will change all of the weights onto **`SThcells[0]`** to 0.6.

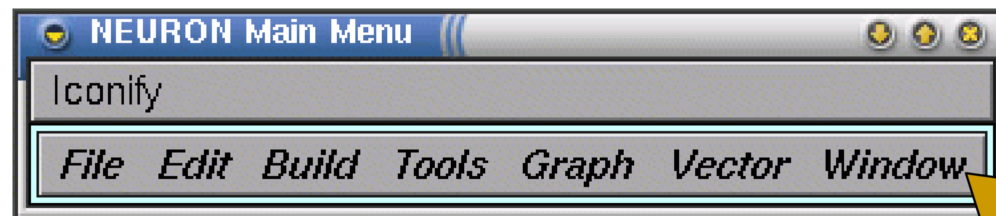
Working with NEURON: practical hints (tut E)

- Quite often we want to analyse or record certain simulation results and perform large numbers of simulations as fast as possible (for example in parameter searching).
- Thus, we will explore methods of getting data out of NEURON to be stored, visualised or analysed in other packages.
- We will also consider ways of speeding up the simulations and the consequences and decision we take in doing this.
- Much more can be found at: http://www.neuron.yale.edu/neuron/ by going to *Documentation* and/or *Programmer's Reference*.

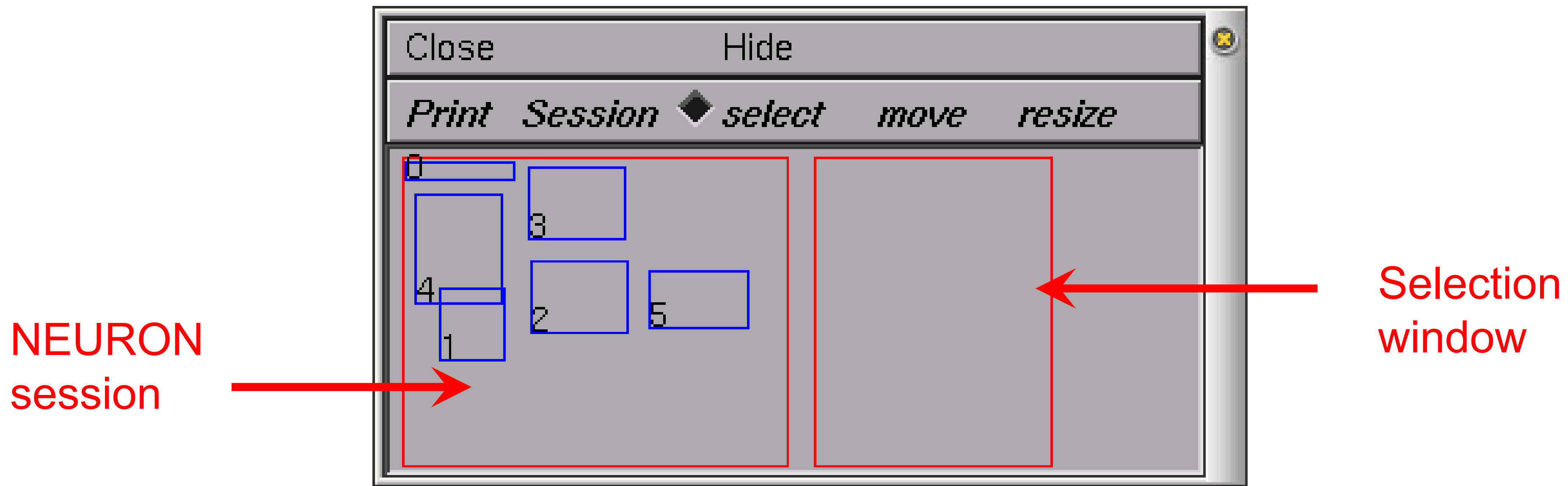
The print & file window manager

- This function enables you to
  - print selected windows from your simulation and also
  - to store the whole session, so that the next time you can continue where you have stopped.

- To open the *Print & File Window Manager*, select *Print & File Window Manager* from the *Window* menu on the *Main menu*.



The print & file window manager



- The left most of the two red rectangles in this window represents the entire NEURON display. Each smaller blue rectangle (with a number in it) represents one of NEURON's windows.
- The second red rectangle represents a sheet of paper--we will call this the Selection rectangle. It is used to print selected windows to a file or printer and to save selected windows in a session file.

Saving and retrieving sessions

- After creating several graphs, you may want to save the windows you have created (i.e., graphs and panels) to a file so that you can recall them at a later time.

- NEURON allows you to save either all or selected windows to a session by selecting the *Save selected* or *Save all* option of the Session menu in the *Print & File Window Manager*.

- Save all will save the position and contents of all NEURON's windows. Save selected will save only those windows that are currently selected in the Selection rectangle in *Print & File Window Manager*. Either of these options will pop up a window, in which you can enter the filename of your saved session.

Saving and retrieving sessions

- If we save our session to a file (e.g., **`sthB.ses`**), we can
  - either load the session each time we load our program by selecting the *Retrieve* option of the *Session* menu in the *Print & File Window Manager*,
  - or we can have our program automatically load our session for us. To do this, we need to add the following at the very end of our program:

```
xopen("sthB.ses")
```

  - where **`sthB.ses`** is the name of the session we saved. The next time we start our program, the session with our graphs and menus will automatically be loaded into NEURON.

Recording data with vectors (w/o plotting them)

- It is often convenient to save the data for later re-use. This can be done in the **`hoc`** file using **`Vector`** and **`File`** objects.

- Let us record both the time and voltage at the soma of the 3rd model neuron. Thus, we create 2 variables for holding the vector objects:

```
objref rect, recv
```

- Now the vectors must be created. We do this with the new command:

```
rect = new Vector()
recv = new Vector()
```

- We now have two vector objects.

Recording voltage

- Our subthalamic cells are defined as an array of object variables (from **`SThcells[0]`** to **`SThcells[nSThcells]`**). Thus the voltage at the soma of the 3rd model neuron is given by this variable:

```
SThcells[2].soma.v(0.5)
```

- To record this voltage, we prepend an "&" to this variable name, and give it as an argument to the record function of the particular vector object where we want it saved, i.e.:

```
recv.record(&SThcells[2].soma.v(0.5))
```

- The voltage in the centre of the 3rd cell's soma will be recorded into this vector on each simulation. **Note**: if a second simulation is run, it will overwrite any previous simulation data in this vector !

Recording time

- To record the time information, the second vector object (**`rect`**) can be setup to record the time variable **`t`**, using this command:

```
rect.record(&t)
```

- Now, when we run a simulation, the time and voltage data from these variables will be recorded in our two vectors (Note: "resize_chunk" messages in the terminal just indicates the vector objects are growing).

- To see the data in the vectors, we can use the vector object function **`printf`** to display the data in the terminal, i.e.:

```
recv.printf()
```

Saving the vectors of data to a file

- One way of doing this is to save each vector to a different file.
- To do this we must create file objects for each file. First a variable must be defined for the object (we will call ours **savv** and **savt**).

```
objref savv, savt
```

- We now create the file objects using the new command:

```
savv = new File()
savt = new File()
```

Opening, writing to and closing a file

- The function **`wopen`** in the file object opens a file for writing to it. It takes as an argument the name of the file, i.e.:

```
savv.wopen("cell3somav.dat")
savt.wopen("cell3somat.dat")
```

- We can now save the data in our vectors (**`rect`** and **`recv`**) using, the **`printf`** function with the file object as an argument:

```
recv.printf(savv)
rect.printf(savt)
```

- Finally, don't forget to close the files:

```
savv.close()
savt.close()
```

Saving the vectors in one file

- First, we create a new single file object:

```
objref savdata
```

- Then we'll open the file, e.g. "**`cell3soma.dat`**", for writing to it:

```
savdata = new File()
savdata.wopen("cell3soma.dat")
```

- We can also save useful information at the beginning of the file, such as the variable name (e.g. **`SThcells[2].soma.v(0.5)`**), and the number of data vectors in the file. We can write this to the file using the **`printf`** function in the File object:

The **`printf`** function in a file object

```
savdata.printf("t SThcells[2].soma.v(0.5)\n")
```

- The **`printf`** function in a File object is somewhat different from the **`printf`** in a Vector.
- As it is used above, it has only one argument, which is the text that will be printed to the file header. This text is enclosed in quotes **"."** and in programming terms is known as a string.
- The **"\n"** at the end of the string inserts a newline.

The **`printf`** function in a file object

```
savdata.printf("%d\n",rect.size())
```

- The second time **`printf`** is used it has two arguments. The first argument is a string, but this time it contains a percentage sign followed by a single character, the letter **`"d"`**. This **`"%d"`** is a placeholder for an integer number that is given to the function as an additional argument. The argument in this example is **`rect.size()`**.

- The function **`size`** is in the Vector object and returns the size of the vector, here, the size of the time vector **`rect`**.

Saving the vectors in one file

- Now we want each line of our file to have two numbers (time and **STh cells[2].soma.v(0.5)** at that time). We use the **for** loop:

```
for i=0,rect.size()-1 {
savdata.printf("%g %g\n",rect.x(i),recv.x(i)) }
```

  - **%g** is a printing convention for real (floating point) numbers. These real numbers are the data from the time and voltage vectors.

- The **x** function in the Vector object returns the data at a given location in the vector. For example, **rect.x(0)**, is the first element in the **rect** vector, **rect.x(1)**, is the second element, etc. In our for loop, we use **x(i)** to cycle through each vector element, so that each data pair gets written to the file in sequence. Finally, we close the file:

```
savdata.close()
```