**create_neurons.py**

**Purpose:** This python script is used to make a final netlist file for a single neuron containing the subcircuit definitions for all the components of a neuron (synapses, voltage adders, buffers, and axon) and also the instantiations of all the subcircuits.

**Method:** This python script is run through the terminal and not a compiler software because that way it allows arguments to be passed to the script. We pass two arguments: the first argument is the complete path of the directory which contains all the input subcircuit/dummy files used by the creating_netlist.py to extract the subcircuit definitions for all the components, and the second argument is the name of the text file which contains the connection table.

The typical bash command we use to run this script with arguments is:

Linux users: **python create_neurons.py home/sidharth/Desktop/neuron_test/basic Neuronlist.txt**
Windows users: **python create_neurons.py C:\somefolder\dummyfiles Neuronlist.txt**

Here the directory path is the first argument and Neuronlist.txt is the second argument. The first argument is different for linux and windows only because of the difference in the file system architecture used by linux and windows.

The script is divided into two parts. The first part is to create an output netlist file for the neuron's netlist. The second part of the code is to make a graph and its visualization of how the internal components of the neuron are connected, save it in an image file and also create an output graph file in the .gexf format. The .gexf format can be used to load the neuron graph in Gephi, which is a Jython based graph visualization and data manipulation software. It has versatile visualization options and also lets you edit the created nodes and edges of the graph. It can also be used to run algorithms on a data set represented as nodes and edges (a graph). Therefore storing the created graph in .gexf format would be very useful for the purpose of debugging, as through graph visualization, we can constantly keep checking if the connections in the circuit are the way they are supposed to be.

**Part I : Netlist creation**
We import NLI_main.py in this code provides access to the Neuron class, which has all the connection information for a neuron organised in the following format:

**Neuron name: C**
**Inputs: [ ‘A’ , ‘B’, ‘H’]**
**types: [ ‘e’ , ‘e’, ‘i’]**
**weights: [ ‘1’ , ‘3’, ‘2’]**

NLI_main.py also provides the subcircuit instantiation statements in the form of lists called synapseini, adderini, etc.

We pass the second argument (name of the connections file) to NLI_main as it uses that data to make the Neuron class and the instantiation statements. One mistake we made here was while trying to further pass the first argument passed into creating_netlist.py into NLI_main. We used netargs() function to store the arguments in a list called stringx.

```
def netargs():
stringx = []
stringx.append(sys.argv[1])
stringx.append(sys.argv[2])
return stringx
```

Now to pass the first argument into NLI_main we used “from create_neurons import netargs” to import the netargs function to get the arguments and then call the netargs function to get the arguments in a list. This did cause an error because it tried to import a creating_netlist.py function while creating_netlist.py was already being compiled and run. To solve this problem we used this :

```
os.system("python NLI_main.py %s"%argu[0])
```

Instead of calling creating_netlist's function from within NLI, we executed the NLI_main code from within the creating_netlist code and passed the first argument into it, thus giving the desired result. For this to work we also had to add the netargs function definition in the NLI_main script too.

In the code we append all the data that has to be written into the output file to **array1** which is later written into the output netlist file once we have all the data in array1.We used functions like **get_adder, get_synapse, get_axon** and **get_buffer** to append the subcircuit definition from the dummy files of adder,axon,synapses and buffer into array1.

For get_synapse, we only add the definitions of those synapses that are used in the neuron. We check from the Neuron class to know what type and weights of synapses are used in the neuron circuit. The format of the synapse dummy file is :

**synapse_type_weight**

As an example, synapse_e_2 would mean the synapse is excitatory and the weight is 2.
There is also function there called **make_neuron**. It creates directories for each neuron with the directory name same as name of the neuron, stores a netlist file inside the directory of that neuron. The file is called netlist. This function will be used later when joining different neurons to make a larger circuit.

The function **get_instantiations** is used to append the instantiation statements into array1. The statements are stored in lists called **adderini, synapseini** and string **axonini.**
With array1 containing all the subcircuit definitions and instantiation statements, which is the complete netlist, we write array1 into an output netlist file.

**Part II : Graph Creation and Gephi**

This part of the code takes the connection data from the input connection data file and creates a graph out according to the connection data and also creates a .gexf format file. .gexf is a gephi compatible format. So the graph created here can be opened and used with gephi and all its data visualization and manipulation features

**add_edge()** and **add_node()** functions of the **networkx** library are used to create edges and nodes of the the graph. The graph is then plot using library called **matplotlib.pyplot**. **nx.savefig()** is used to save an image file of the graph. **nx.write_gexf()** is used to save the graph in a .gexf format.

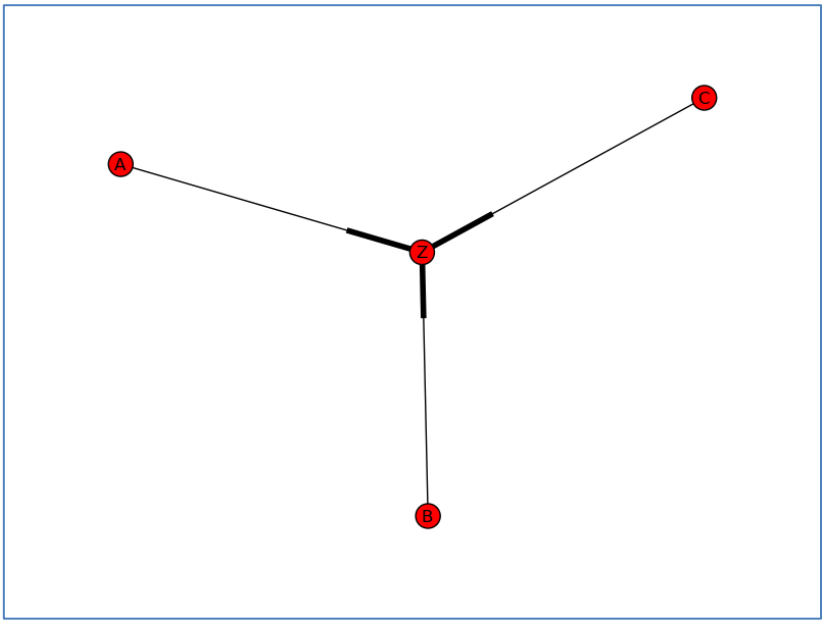


**Illustration 1: This is what a typical GEPHI graph looks like**

**Version of Python used** : Python 2.7.3

**Python Libraries used :**

1. **os** : in-built function os.system is used to
2. **sys :** sys.argv is used to return arguments passed from the terminal
3. **subprocess :** subprocess.call() used to run bash commands
4. **networkx :** used for making graph and saving as .gexf
5. **matplotlib.pyplot:** used for plotting a graph out of the data of networkx

## Code

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 4 05:18:43 2013
Addition from version 1.: Putting array1 into a final netlist file
@author: siddharth
"""
import os
import sys
#function to store the arguments passed through the command line into
#a list called stringx
def netargs():
stringx = []
for ele in sys.argv:
stringx.append(ele)
#stringx.append(sys.argv[1])
#stringx.append(sys.argv[2])
return stringx
```

```
argu = []
argu = netargs()

array1 = []
#NLI_string = ''
#for ele in argu:
# NLI_string += ele + ' '



#NLI_main contains the Neuron class and instantiation lists used in this code
raw_input()
from NLI_main import *
import subprocess

#function to append the contents of the axon definition file into
#the array1 which would be written into the file
def get_axon(filename):
print "------------------------------------------------------------"
f1 = open(filename,"r")
for line in f1:
array1.append(line)
array1.append('\n')
f1.close()
#calling the get_axon function by passing the complete path of the file
#the directory is taken as the second argument passed through the command prompt
get_axon("%s/axon" % argu[1])
#function to append the contents of the adder definition file into the
#array1 which is a list that contains all data that would be written into
#the output netlist file
def get_adder(filename):
f1 = open(filename,"r")
for line in f1:
array1.append(line)
array1.append('\n')
f1.close()

#calling the get_adder function
get_adder("%s/voltage_adder"% argu[1])



#function to take the buffer circuit definition and append it into
#array1.
def get_buffer(filename):
f1 = open(filename,"r")
for line in f1:
array1.append(line)
array1.append("\n")
f1.close()
```

```
#calling get_buffer with complete file path. again, directory is passed
#in the arguments through the command line
get_buffer("%s/buffer" % argu[1])

#function to get the circuit definition from the dummy circuit definition file for synapse
#and append into array1
def get_synapse(neuron):
namearray = []
for i in range(len(neuron.inputs)):
name = argu[1] +"/synapse_"+ neuron.types[i] + "_" + neuron.weights[i]
if name not in namearray:
namearray.append(name)
for ele in namearray:
f1 = open(ele,"r")
for line in f1:
array1.append(line)
array1.append('\n')
#f1.close()
#calling the get_synapse function for all synapse types and names
#taken from neuron class
for i in range(len(Neuron)):
get_synapse(Neuron[i])

#for line in array1:
# print line

#function to create directories for each neuron, directory name same as name
#of the neuron, store a netlist file inside the directory of that neuron.file is called
#netlist
def make_neuron(neuron):
name = neuron.name
str1 = "
print "-----------------------------"
str1 = "mkdir " + argu[1] +'/' + name
#print str1
strw = argu[1] + '/' + name
subprocess.call(str1,shell=True)
filename= strw + "/netlist"
f1 = open(filename,"w")
line1 = "// Library name: C_elegan\n"
line2 = "// Cell name:" + name + "\n"
line3 = "// View name: schematic\n"
str2 = ""
for i in range (len(neuron.inputs)):
str2 += " " + neuron.inputs[i]
line4 = "subckt " + name + str2 + " output ""\n"
line5 = "ends " + name + "\n"
line6 = "// End of subcircuit definition.\n"
lines = line1 + line2 + line3+line4+line5+line6
#print lines
```

```
f1.writelines(lines)
f1.close()

#calling make_neuron function for every Neuron class object i.e.
# a neuron. making directory for each neuron, which contains its netlist file
for i in range(len(Neuron)):
make_neuron(Neuron[i])

#creating and opening netlist file for writing array1 into.
f1 = open("netlist","w+")
#print array1

#function to get all the instantiations of all the neuron circuit components:
# synapses, adders, buffers and axon.
#appends the instantiation statements to array1 to be written into the o/p file
def get_instantiations():
array1.append("// Library name: C_elegan")
array1.append("\n")
array1.append("// Cell name: D\n")
array1.append("// View name: schematic")
for ele in adderini:
for i in range(len(ele)):
array1.append("\n")
array1.append(ele[i])
for ele in synapseini:
for i in range(len(ele)):
array1.append("\n")
array1.append(ele[i])
#array1.append("\n")
array1.append("\n")
# print axonini
array1.append(axonini)
array1.append("\n")

#function call
get_instantiations()


for line in array1:
f1.writelines(line)
print line
'''
```

GEPHI

This part of the code takes the connection data from the input connection data file and creates a graph out according to the connection data and also creates a .gexf format file. .gexf is a gephi compatible format. So the graph created here can be opened and used with gephi and all its data visualization and manipulation features

'''

#Used networkx to make the graph: nodes and edges
#used matplotlib.pyplot to draw graph
import matplotlib.pyplot as plt
import networkx as nx
g = nx.MultiDiGraph()#Create graph object, with multi directed graph
#g = nx.sedgewick_maze_graph()
#g = nx.complete_graph(5)
#g = nx.tetrahedral_graph()
#g = nx.petersen_graph()

array_gephi = []

#adding nodes and edges using networkx
for i in range(len(Neuron)):
g.add_node(Neuron[i].name)
if Neuron[i].inputs:
for k in range(len(Neuron[i].inputs)):
g.add_edge(Neuron[i].inputs[k],Neuron[i].name)
g.get_edge_data(Neuron[i].inputs[k],Neuron[i].name)

#g.add_edge(Neuron[1].inputs[0],Neuron[1].name)
#g.[A][C]['color'] = 'blue'
#nx.write_gml(g,"test.gml")
nx.write_gexf(g, "sid.gexf")#saving as gexf file
nx.draw(g)

plt.savefig("simple_graph.png")
plt.show()#plots graph

## NLI_main.py

**Purpose:** The NLI_main.py code, or Net List Instantiation code, creates all of the instantiations for synapses, adders, buffers, and axon circuits. It is written in python version 2.7.3.

**Method:** This code is called by the create_neurons.py (We'll call this CN). The CN code passes a single argument to the NLI code. The argument is the name of a text file which contains a list of neurons, their connections, and the type of connections. From this list, NLI creates class objects that store the information of each neuron. The code then uses this information to decide how many adders and synapses each neuron needs as well as what inputs and outputs every adder, buffer, and synapse should have. The code will then store the circuit instantiations into three different lists; adderini, synapseini, and axonini. These lists are then used in the CN program where it is inputted into the netlist file.

**Grabbing the information**
The first part of the code reads the file that was passed to NLI. The following image is a sample of a neuron list with neuron, C, which has three inputs; A, B, and H. It also contains

the weight of the synapse and its type. Note that C must have a dummy output in order to avoid errors.

```
Neuron Target Weight Type
A C 1 e
B C 3 e
H C 2 i
C x - -
```

The program goes through the text file and saves the name of the neurons, which neurons are targeting it and the type of synapse. This information is then saved into a class object in the following format. Each column has the information for a single input. For example, Input A is excitatory with a weight of 1

```
Neuron name: C
Inputs: [ 'A' , 'B', 'H']
types: [ 'e' , 'e', 'i']
weights: [ '1' , '3', '2']
```

**Creating the Adders/Buffers Instantiations**

The next part of the code creates and adder/buffer network, and decides which inputs and outputs they should have. The difficulty of this portion of the code is making sure that each input is properly added no matter how many inputs there are. We can see this using a simple three input neuron. All of the inputs must be added however, because we are using 2-input adders we sometimes have an odd input that must wait to be added. In this case, net1 and net2 are added first. Net3 is sent to a buffer so that it does not reach Adder2 before the output of Adder1 can reach it.


Using an algorithm we are able to store the inputs and outputs for each adder and buffer into a two dimensional list. Each component, adder or buffer, is saved as a list which is saved into a master list. In this case, we would have a master list that looks like this: [ [1,2,4], [3,5], [4,5,6] ]. This first element [1,2,4] shows that an adder, we know it's an adder because it has three elements, has inputs 1 and 2 with an output named 4. The second element is a buffer; we know it is a buffer because it has two elements inside of it. It has in input 3 and an output 5. At this stage we have all of the information we need in order to create the netlist instantiations for the adders and buffers.

Using a ‘for loop’ we create an instantiation for each component using the following format for adders and buffers where n is the nth component that has been created. We know to use the buffer format when the list element contains only 2 elements.

I’n’ (input input output) Voltage_adder
I’n’(input output) buffer

Adder1 would be

I0 (net1 net2 net4) Voltage_adder

Buffer1 would be

I1 (net3 net5) buffer

Once all of the instantiations have been completed, they are stored in a list, adderini.

**Creating the Synapse Instantiations**

For this section of the code we had to use our neuron class objects. By analyzing the inputs of the adders and the Neuron we can determine how to name the synapses. Each input to the neuron would be the input to a synapse. Additionaly, each input to the adder/buffer network would be an output for a synapse. However; because we know that the adder/buffer networks have the same amount of inputs as each neuron we were able to simply count the number of inputs the neuron had an associate each neuron input with a number. For example;

| |
|---|
| Neuron name: C<br>Inputs: [ ‘A’ , ‘B’, ‘H’]<br>types: [ ‘e’ , ‘e’, ‘i’]<br>weights: [ ‘1’ , ‘3’, ‘2’] |

we can say that A is the 1st input, B is the 2nd, and H is the 3rd. we then save this information into another list of lists. [ [A,1], [B,2], [C,3] ]. This is almost all of the information we need in order to create the instantiations for the synapses. One thing we had to take account of was the number of instantiations we had created so we never named an instantiation with the same name as another one. That is to say, if we had created adders, I0 and I2, in addition to buffer I3 then we had to start naming our synapses with I4, I5... I’n’. In order to do this we counted the number of elements in our adderini list and started with a number over that count. The format for these synapses were;

I’n’ (input output) synapse_’type’_’weight’

The synapse instantiation for input A would be

I4 (A net1) synapse_2_1

Lastly, we create the axon instantiation. For the axon, we simply take the last output from the last adder and assign it as the input for the axon. We also had to remember to name the axon using the counting method described above. The axon is formated as follows along with an instantiation we would use for the example we have been looking at

I’n’ (input output) axon
I7 (net6 output) axon

**To finalize the code, we save the synapses into synapseini and the axon into axonini. These 3 lists are then sent back to the CN program for use.**

## Code

```
# -*- coding: utf-8 -*-
"""
NLI_main.py - 7/22/2013 - Python version No. 2.7.5
Author: Chase Cook - Contributions: Siddarth, Akshay
Objective: The NLI_main.py code, or Net List Instantiation code,
creates all of the instantiations for synapses, adders, buffers, and
axon circuits.

"""

class neuron:
Ncnt = 0
# constructor
def __init__(self, name, inputs, types, weights):
self.name = name
self.inputs = inputs
self.types = types
self.weights = weights
neuron.Ncnt += 1
# Displays all attributes of an object
def display_attribs(self):
print 'Neuron Name: ',self.name
print 'Inputs: ',self.inputs
print 'Type : ',self.types
print 'Weight: ',self.weights
class adder:
addcnt = 0
# constructor
def __init__(self, name, inputs, output):
self.name = name
self.inputs = inputs
self.output = output
adder.addcnt += 1
# Displays all attributes of an object
def display_attribs(self):
print 'Adder Name: ',self.name
print 'Inputs: ',self.inputs
print 'Output : ',self.output
class buff:
def __init__(self,name,inp,output):
self.name = name
self.inp = inp
```

```
self.output = output
def display_attribs(self):
print 'Buffer Name: ',self.name
print 'Input: ',self.inp
print 'Output: ',self.output
# Retrieves the specified files that is passed to the function
def get_file(filename):
array =[]
if type(filename) == list:
array = make_file(filename)
return array
fp = open(filename,"r")
for line in fp:
array.append(line.split())
print array
fp.close()
return array
def make_file(inlist):
arr = []
arr.append('')
for ele in inlist:
ar = ele.split('_')
arr.append(ar)
return arr
# Displays an array
def display_buffer(array):
for line in array:
print line

# retrieves the names of the neurons from the neuron list
def get_names(array):
names = []
print array
for element in array:
print element[0]
if element[0] in names:
pass
else:
names.append(element[0])
print names
return names
# retrieves the attributes of the passes neuron name.
def get_attributes(name):
i = 0
#Arrays to store respective attributes
inputarr = []
weightsarr = []
typesarr = []
attribtmparr = []
#print arr
```

```
while i < len(connections):
# variables to check and store the attribute before entering into array
nametmp = connections[i][1]
intmp = connections[i][0]
weighttmp = connections[i][2]
typetmp = connections[i][3]
# checks if the attribute is associated with the passed neuron
# and saves it if it is.
if nametmp == name:
inputarr.append(intmp)
weightsarr.append(weighttmp)
typesarr.append(typetmp)
# does nothing if attribute is not associated with passed neuron
else:
pass
# while loop control variable
i += 1
#saves the arrays into one list
attribtmparr.append(inputarr)
attribtmparr.append(weightsarr)
attribtmparr.append(typesarr)
# returns the list with all attributes of passes neuron
return attribtmparr
def make_objects(nrn):
for ele in nrn:
attributes.append(get_attributes(ele))
inputsar = []
weightsar = []
typesar = []
for line in attributes:
inputsar.append(line[0])
weightsar.append(line[1])
typesar.append(line[2])

i = 0
while i < len(nrn):
nrn[i] = neuron(nrn[i],inputsar[i],typesar[i],weightsar[i])
i += 1
return nrn
def get_syn_ini():
fin_ini=[]
comp_counter = []
for i in ConnectionList:
comp_counter.append(len(i))
cmp_index = 0
for ele in wrkinNrns:
tmp_ini = []
Ncnt = 0
incnt = ele[1]
for i in range(incnt):
```

```
nametmp = 'I'+str(Ncnt+comp_counter[cmp_index])+' ('+Neuron[ele[2]].inputs[i]+'
net'+str(i+1)+') synapse_'+Neuron[ele[2]].types[i]+'_'+Neuron[ele[2]].weights[i]
Ncnt += 1
tmp_ini.append(nametmp)
cmp_index += 1
fin_ini.append(tmp_ini)
return fin_ini
def get_ax_ini(cnt):
tmp = ''
tmp_ini = []
comp_index = 0
for i in range(len(synapseini)):
tmp = 'I'+str(len(synapseini[i])+len(adderini[i]))+' (net'+str(outputs[i])+' output) axon'
comp_index += 1
tmp_ini.append(tmp)
return tmp_ini
def get_out():
outs = []
for ele in ConnectionList:
outs.append(ele[len(ele)-1][2])
return outs
def getInterNets(inter_nets):
arr=[]
tmp=[]
for i in nets:
for j in i:
tmp.extend(j)
arr.append(tmp)
tmp=[]
last_set=len(nets)-1
last_set_e=len(nets[last_set])-1
new_e=nets[last_set][last_set_e][1]+1
tmp.insert(0,new_e)
arr.append(tmp)
return arr


def reverse(inter_nets):
super_pairs=[]
for i in inter_nets:
bit=0
pairs=[]
tmp=[]
for j in range(len(i)):
x=[]
x.insert(0,i[j])
tmp.extend(x)
if j%2==1 or j==len(i)-1:
pairs.append(tmp)
tmp=[]
```

```
if bit==0:
bit=1
else:
bit=0
super_pairs.append(pairs)
pairs=[]
return super_pairs

def getNetConnections(nets):
connects=[]
tmp=[]
for i in range(len(nets)):
inst=[]
for j in range(len(nets[i])):
x=[]
tmp.extend(nets[i][j])
x.insert(0,inter_nets[i+1][j])
tmp.extend(x)
inst.append(tmp)
x=[]
tmp=[]
connects.extend(inst)
inst=[]
tmp=[]
return connects
def make_ini(lst):
tmpList = []
for ele in lst:
tmpLine = []
cnt = 0
for i in ele:
tmp = ''
if len(i) == 3:
tmp = 'I'+str(cnt)+' (net'+str(i[0])+' net'+str(i[1])+' net'+str(i[2])+') Voltage_adder'
else:
tmp = 'I'+str(cnt)+' (net'+str(i[0])+' net'+str(i[1])+') buffer'
tmpLine.append(tmp)
cnt +=1
tmpList.append(tmpLine)
return tmpList
def find_nrns():
tmp1 = []
counter = 0
for ele in Neuron:
cnt = len(ele.inputs)
tmp = []
if cnt == 0:
pass
else:
tmp.append(ele.name)
```

```
tmp.append(len(ele.inputs))
tmp.append(counter)
tmp1.append(tmp)
counter +=1
return tmp1
def get_tree(n):
TreeCells = []
tmp = []
while n != 1:
tmp = []
if n%2 == 0:
tmp.append(n/2)
tmp.append(0)
TreeCells.append(tmp)
n = n/2
else:
tmp.append((n-1)/2)
tmp.append(1)
TreeCells.append(tmp)
n = (n-1)/2 + 1
return TreeCells
def get_sum():
numlist = []
cnt = 1
for ele in Tree:
tmpnumlist = []
for num in range(2*ele[0]):
tmpnumlist.append(cnt)
cnt+=1
for num in range(ele[1]):
tmpnumlist.append(cnt)
cnt += 1
print tmpnumlist
numlist.append(tmpnumlist)
print cnt
return numlist
def get_args():
pass
'''
MAIN-1: Creating the Neurons
'''

#This loads the arguments that are passed to the program. It can either take
# the file path of a text file that spcifies nueron connections or it can take
# arguments that directly contain the neuron connections.
import sys
def netargs():
if len(sys.argv) > 3:
tmparr = []
for ele in sys.argv:
```

```
tmparr.append(ele)
del tmparr[0]
del tmparr[0]
return tmparr
else:
stringx = ''
stringx = sys.argv[2]
return stringx

# connections is a list that will contain all connection parameters such as neuron name, wiehgt,
# inputs, and type of inputs
connections = []
# argtmp will hold the arguments that are inputed by user
argtmp = ''
# calls a function, netargs, that will retrieve the input arguments
argtmp = netargs()
#loads the list of neurons and their connections
connections = get_file(argtmp)
#gets rid of any headings in the list
del connections[0]
#finds the names of all the neurons
Neuron = get_names(connections)

#array to store attributes
attributes = []
#creates the Neuron class objects
make_objects(Neuron)
'''
MAIN-2: Creating the adders and buffers
'''
#list declarations
Tree = []
inter_nets=[]
connect=[]
ConnectionList = []
axonini = []
#loop will create adders and buffers for all neuron class objects
for ele in Neuron:
cnt = len(ele.inputs)
if cnt == 0: #if Neuron has no inputs, we don't create any adders
pass
else:
# These functons are used to create a naming scheme that will be used
# to create the instantiations for adders and buffers
numbelist= []
Tree = get_tree(cnt)
numberlist = get_sum()
nets=reverse(numberlist)
inter_nets=getInterNets(nets);
```

```
connect=getNetConnections(nets)
if cnt == 0:
pass
else:
ConnectionList.append(connect)
# this list holds all of the instantiations for adders and buffers
adderini = make_ini(ConnectionList)
# this list will hold the neurons to be worked on. the find_nrns function
# will find which neurons require synapses to be made
wrkinNrns = find_nrns()
# this list will hold the instantiations for the synapses. the get_syn_ini
# function will create a synapse for all neurons that need one
outputs = get_out()
synapseini = get_syn_ini()
# this list and function will store the outputs for use as inputs to axon circuits
outputs = get_out()
# this creates the axon instantiations for all axons
axonini = get_ax_ini(outputs)
# Displays the contents of each list to verify that all instantiations have been made
for ele in Neuron:
ele.display_attribs()
for ele in adderini:
print ele
for ele in synapseini:
print ele
for ele in axonini:
print ele
```

## Perl Automation to Create graphical stimuli and input.scs

The perl script automation.pl is used to create the input.scs and graphical stimuli files. After creation of the files, it moves the netlist, input.scs and graphical stimuli files to the 'C' neuron folder.

## Code

```
#!usr/bin/perl

print "Enter the path for the tsmc file \n";

$path = <>;
#$file1="input1.scs";

##########################################################################
########################################
#Creating input.scs"

#Add the header to the input.scs file
```

```
open XYZ, ">input.scs" or die;
print XYZ "simulator lang=spectre\n";
print XYZ "global 0 vss! vdd!\n";
print XYZ "parameters Spread_e=.3 Reuptake_e=1 Weight_e=1.8 E_channel_e=1.5
Pump_e=.3 load=1f\n";
#
print XYZ "include $path\n";

#Open the netlist file
open ABC, "netlist" or die;
#Copy the entire netlist into input.scs
while (<ABC>){
print XYZ $_ ;
}

#Add the footer to the netlist file
print XYZ "include \"./_graphical_stimuli.scs\"\n";
print XYZ "simulatorOptions options reltol=1e-3 vabstol=1e-6 iabstol=1e-12 temp=27 \\
tnom=27 scalem=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 maxwarns=5 \\
digits=5 cols=80 pivrel=1e-3 sensfile=\"../psf/sens.output\" \\
checklimitdest=psf
tran tran stop=20n errpreset=moderate write=\"spectre.ic\" \\
writefinal=\"spectre.fc\" annotate=status maxiters=5
finalTimeOP info what=oppoint where=rawfile
modelParameter info what=models where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
designParamVals info what=parameters where=rawfile
primitives info what=primitives where=rawfile
subckts info what=subckts where=rawfile
saveOptions options save=allpub";

close(XYZ);
#Creating graphical_stimuli.scs

#Add the required headers
open GPH, ">_graphical_stimuli.scs" or die;
print GPH "_vReuptake (Reuptake 0) vsource dc=1.8 type=dc
_vSpread (Spread 0) vsource dc=800n type=dc
_vE_channel (E_channel 0) vsource dc=300n type=dc
_vPump (Pump 0) vsource dc=300n type=dc\n";

#Open the weight_map file of the synapses and add them into the netlist
open MAP, "weight_map.txt" or die;
while(<MAP>){
@arr=split(/ /);
chomp($arr[1]);
print GPH "_vsynapse_$arr[0] (synapse_$arr[0] 0) vsource dc=$arr[1] type=dc\n";
}
```

```
#Check for the number of inputs in the input file
open IPT, "input_count.txt" or die;
while(<IPT>){
@arr1=split(//);
}



$count = scalar(@arr1);

#Add the footer

for($i=0;$i<$count;$i=$i+1){
chomp($arr[$i]);
print GPH "_v$arr1[$i] ($arr1[$i] 0) vsource val0=0 val1=1.8 period=10n delay=150p
rise=300p fall=300p width=5p type=pulse\n";
}

print GPH "_vvss! (vss! 0) vsource dc=-1.8 type=dc
_vvdd! (vdd! 0) vsource dc=1.8 type=dc\n";

system ("mv netlist C");
system ("mv input.scs C");
system ("mv _graphical_stimuli.scs C");
```

## How to run the codes

**Step 1 :**
Put all the 'dummy' files and create_neurons.py, NLI_main.py and automation.pl into the Basic_Files folder. Also, cd into the Basic_Files path in the terminal.

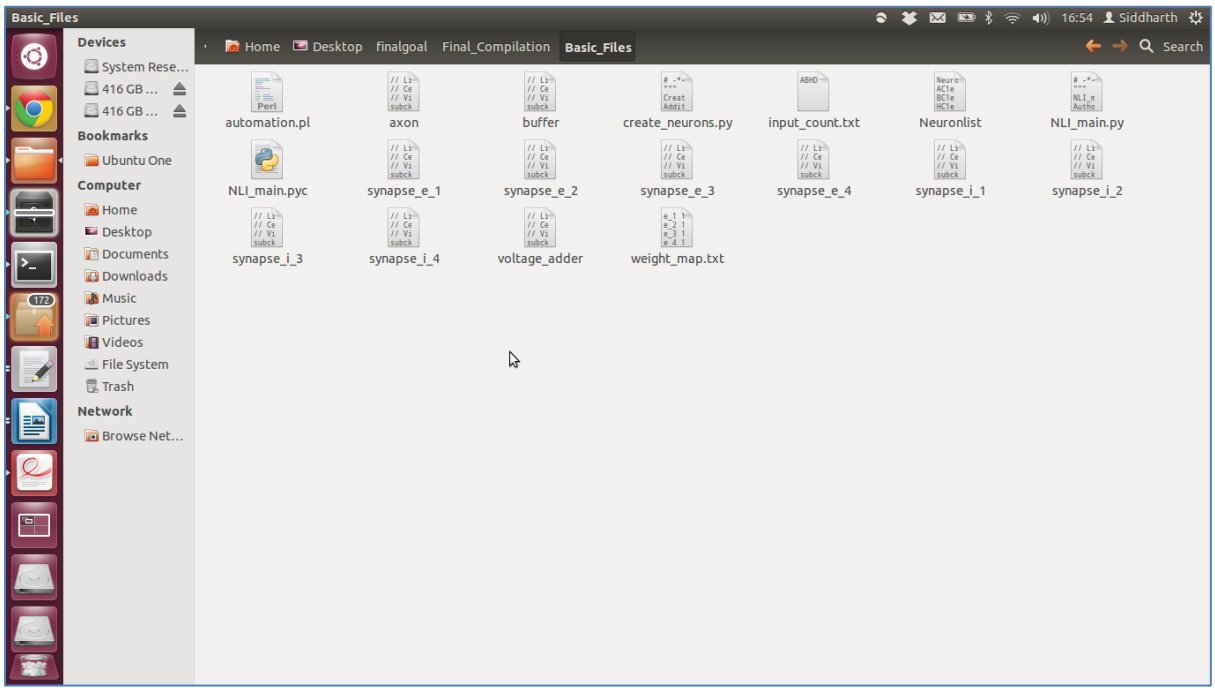


**Illustration 2: Step 1**

**Step 2:**
As shown in Illustration 3, the Neuronlist contains the circuit required by the user in tabular form, and the create_neurons.py creates the netlist for the required circuit.

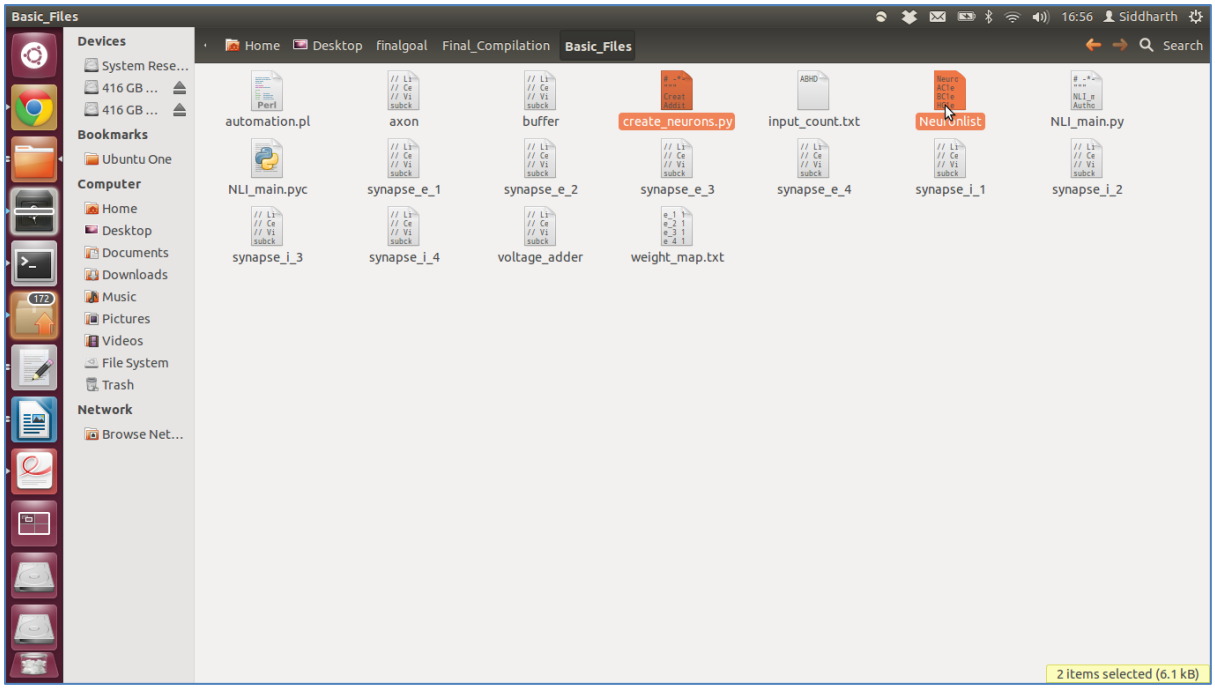


**Illustration 3: Step 2.1**

Now, run the python script create_neurons.py by using the command:

**python create_neurons.py “basic_files path” Neuronlist**

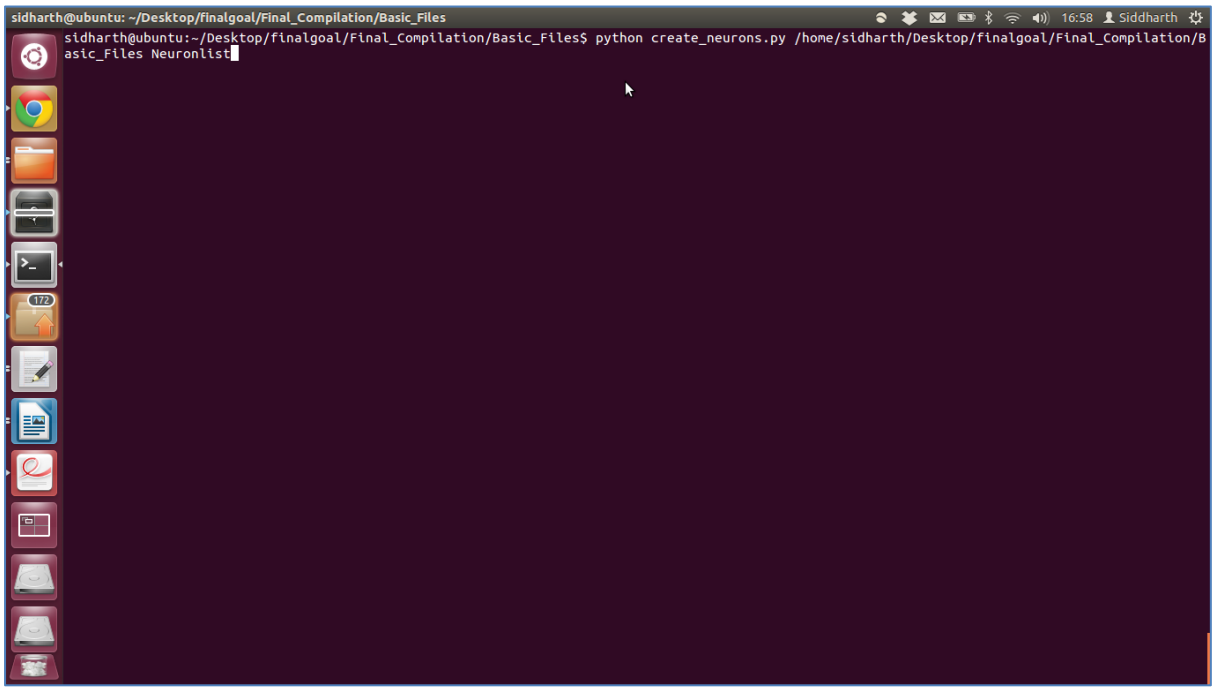


**Illustration 4: Step 2.2**

Once the python code has been executed, you would be able to see the Gephi visualization of the created circuit, as shown in illustration 5.

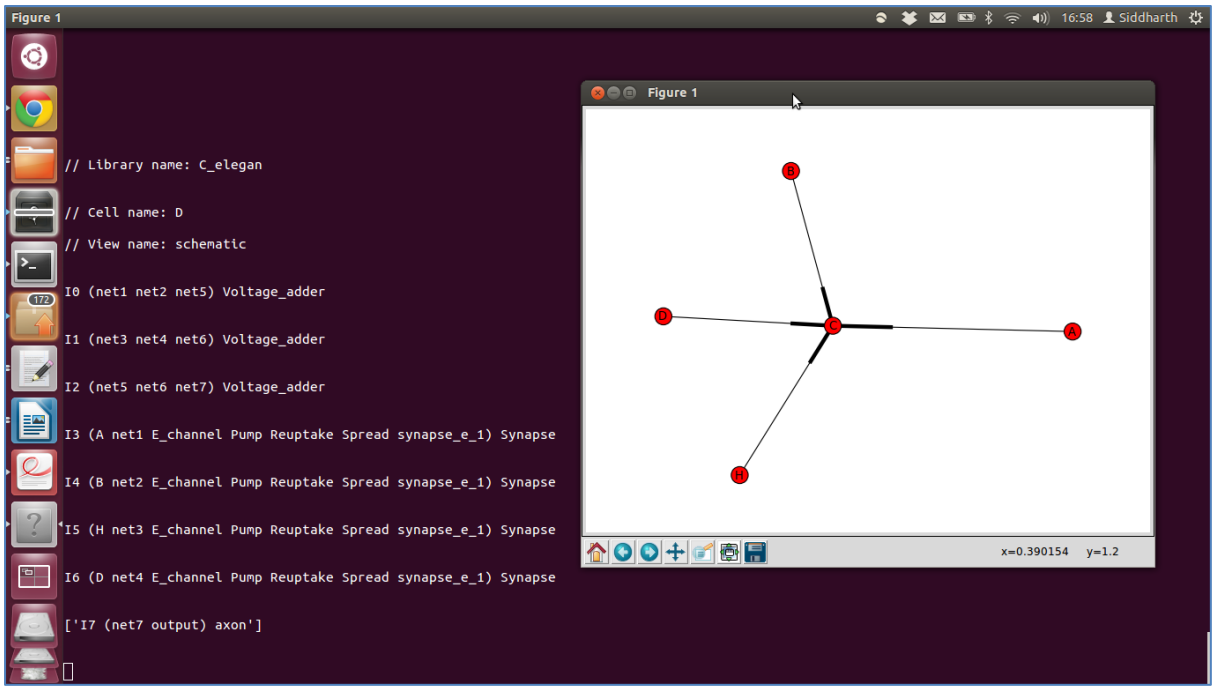


**Illustration 5: Step 2.3**

Also, the netlist file and folders for all the neurons would get created in the Basic_Files folder.

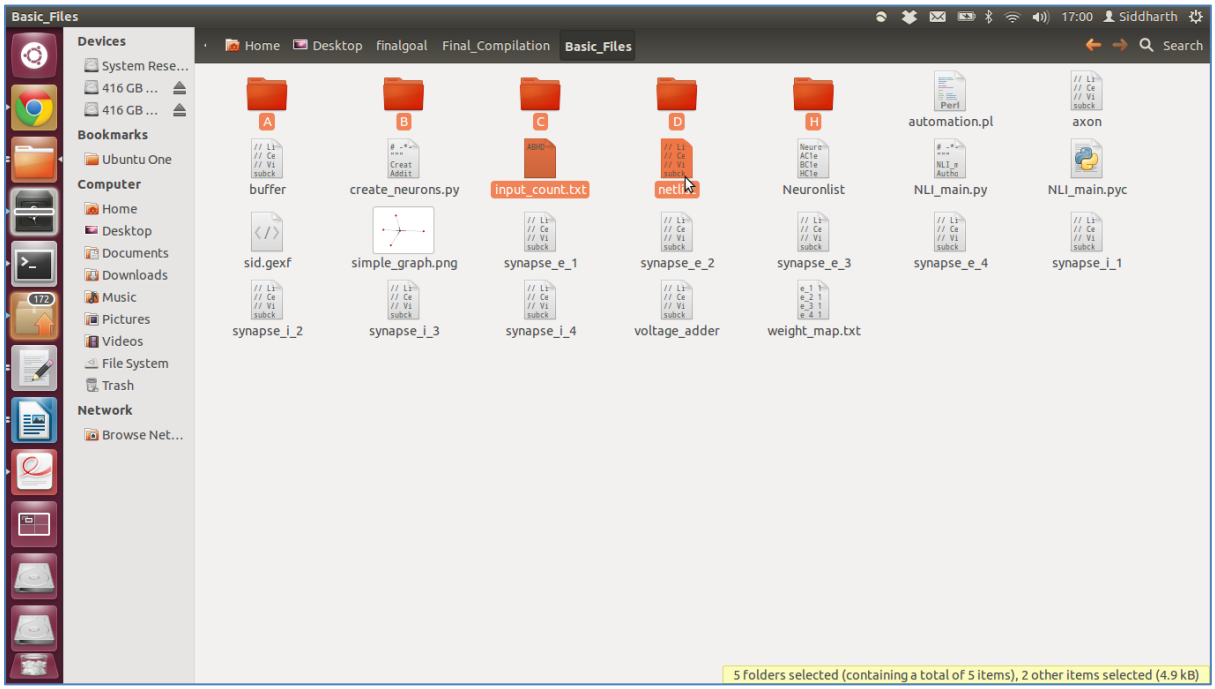


**Illustration 6: Step 2.4**

**Step 3:**
Now we have to run the automation.pl script to create graphical_stimuli and input.scs files. Use the command perl automation.pl for this.

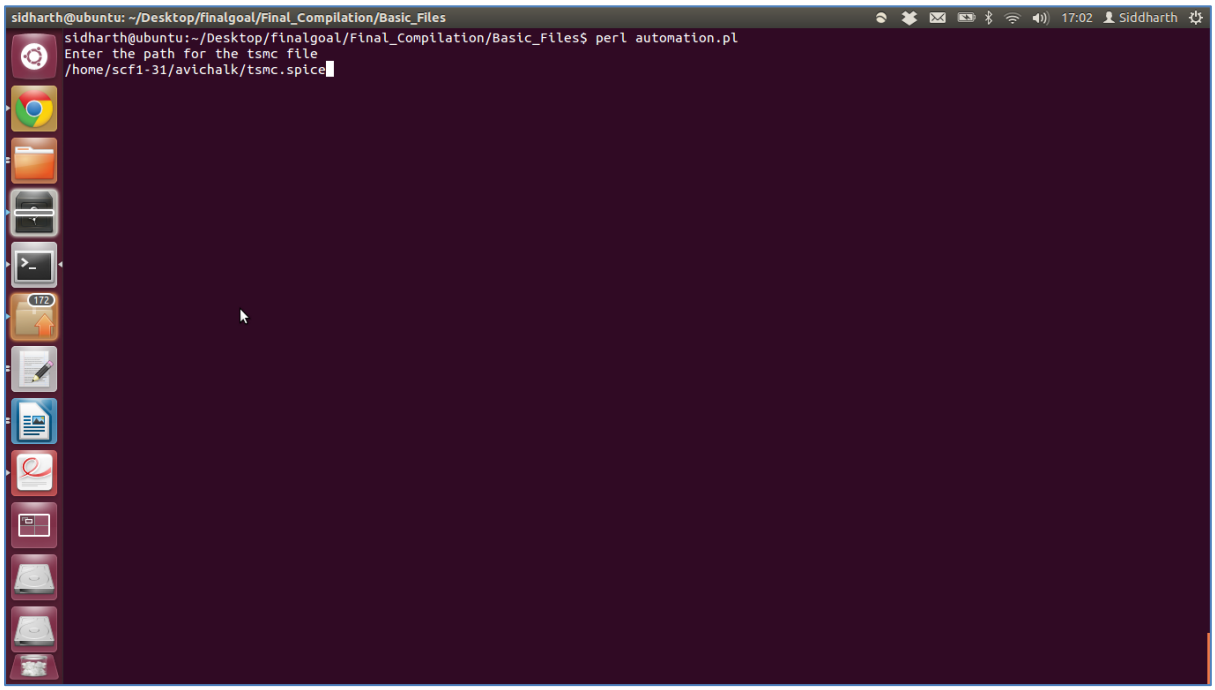


Illustration 7: Step 3.1

After execution, the automation.pl script creates the graphical stimuli and input.scs files. After creation of these two files, it also moves the netlist, input.scs and graphical stimuli files to the C neuron folder. Illustration 8 shows the same.

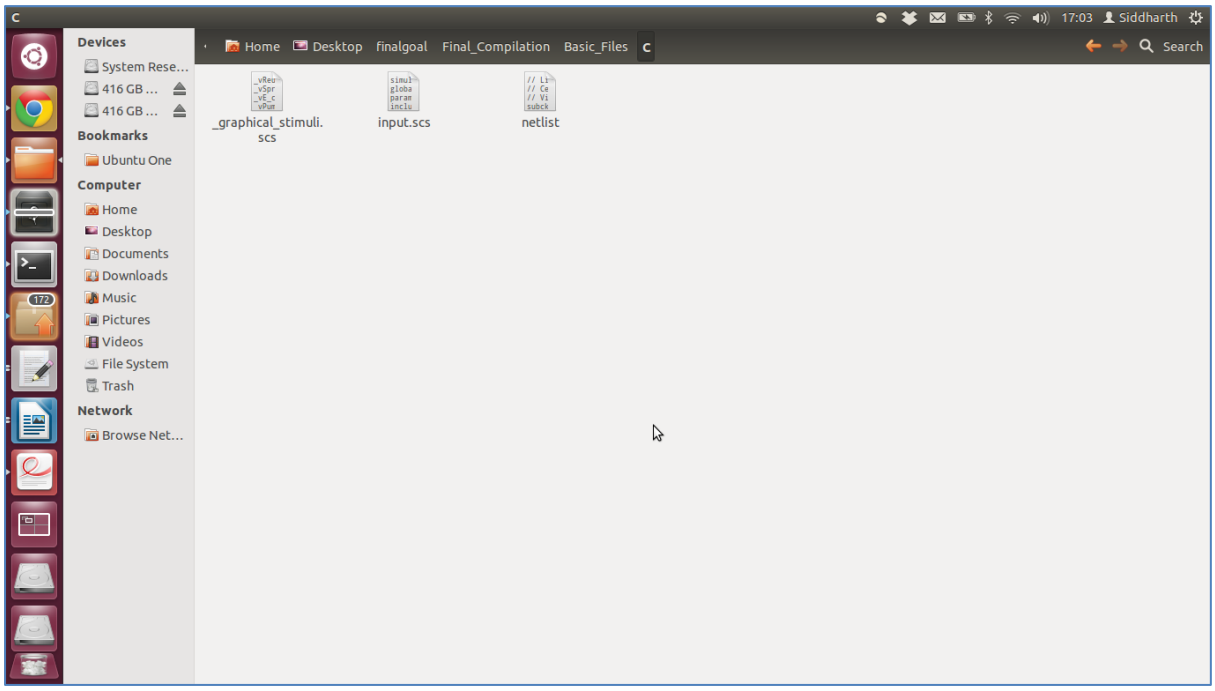


Illustration 8: Step 3.2