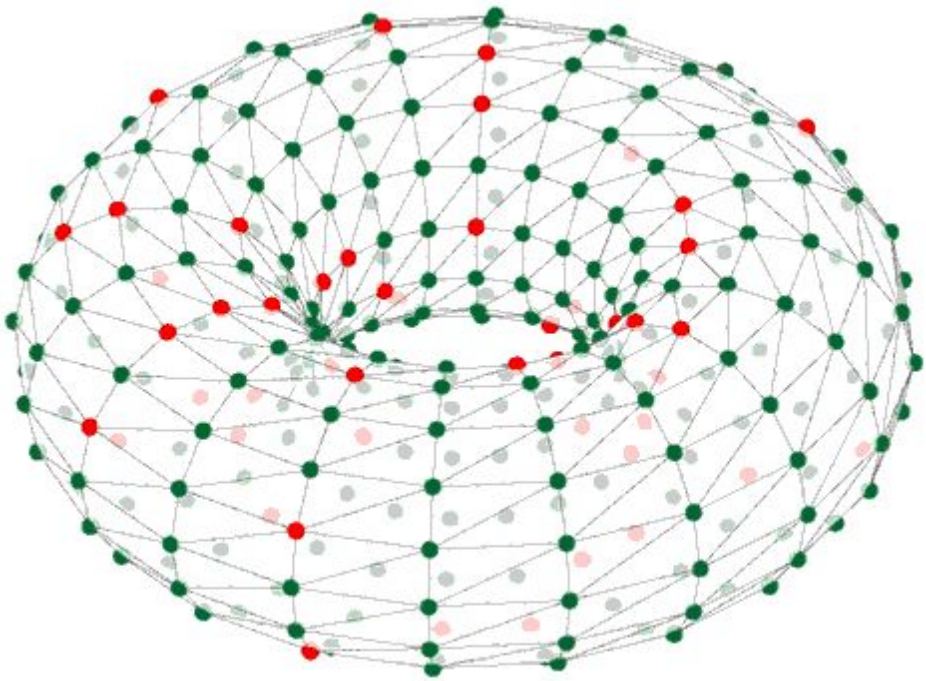
6th SpiNNaker Workshop

Proceedings

September 5th- 9th 2016



Manchester, UK

6th SpiNNaker Workshop

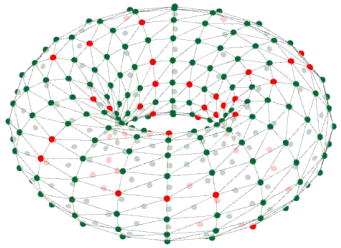
Day 1

September 5th 2016

Time	Session	Presenter
09:00	Registration	
10:00	Workshop Introduction & logistics	SD
10:15	SpiNNaker Hardware & Tools Overview	SD
11:00	Introductory Lab	AGR
12:00	Lunch	
13:00	SpiNNaker architecture and chip resources	ST
14:00	Running PyNN simulations on SpiNNaker	AGR
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Manchester, UK

6th SpiNNaker Workshop



Welcome & Overview

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

Sessions and Venues

- Two types of sessions:
  - Presentations (some optional!)
  - Lab work (with lab books)
- Three venues:
  - ***Collab 1*** (here) for labs and some presentations
  - ***Atlas-1*** (30m away) for some presentations
  - Area outside this room for lunch/drinks breaks
- Don't leave valuables here overnight!

Breakdown of each day

- Sessions:
  - Start at 9am – promptly!
  - Run to 5pm,except the last day (1pm)
- Breaks:
  - Drinks mid-morning and at 3pm daily (half hour)
  - Lunch at 12pm daily (one hour)
- Fire alarm test on Wednesday at 1pm

WI-FI Access

- *eduroam* is available as normal around the building
- UoM guest accounts available if you need one
  - Please ask!

Day 1 - Monday 5th

Time	Session	Presenter
09:00	Registration	
10:00	Workshop Introduction & logistics	SD
10:15	SpiNNaker Hardware & Tools Overview	SD
11:00	Introductory Lab	AGR
12:00	Lunch	
13:00	SpiNNaker architecture and chip resources	ST
14:00	Running PyNN simulations on SpiNNaker	AGR
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Day 2 – Tuesday 6th

Time	Session	Presenter
09:00	SpiNNaker system software (SARK)	ST
10:00	Coffee (earlier than usual)	
10:30	SpiNNaker API + event driven simulation	LAP
11:30	Writing Applications on SpiNNaker - Overview	SD
12:00	Lunch	
13:00	Introduction to Graph Front End (GFE)	ABS (AGR)
14:00	ybug and gdb walk-through	ST
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Day 3 – Wednesday 7th

Time	Session	Presenter
09:00	Simple data I/O and visualisation	ABS (SD)
10:00	Lab time (coffee at 10:30)	
11:00	Maths & fixed point libraries	MH
12:00	Lunch	
13:00	Adding new neuron models	AGR/MH
14:00	Connecting SpiNNaker to external devices	ABS (DRL)
14:30	Lab time (coffee at 15:00)	
16:30	Close	

Day 4 – Thursday 8th

Time	Session	Presenter
09:00	Adding new models of synaptic plasticity	JK
09:45	Graph Front End – further details	ABS (AGR)
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	Using big SpiNNaker machines remotely: The HBP portal	AGR
13:30	Lab time (coffee at 15:00)	
15:30	Demonstration of NENGO language and environment	TBC
16:30	Close	

Day 5 – Friday 9th

Time	Session	Presenter
09:00	Lab time	
10:30	Coffee	
11:00	Lab time	
12:00	Lunch and close	

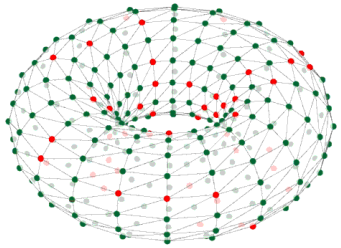
Loan board requests…

- 4-node boards can be loaned out
  - But supply is limited
- Please send an email with your project details to:
  simon.davidson@manchester.ac.uk
- Steve Temple will allocate and log board loans
  - Please don't just take one away….

Feedback…

- We'd appreciate some feedback on…
  - Your workshop experience
  - SpiNNaker hardware
  - SpiNNaker software
  - I'll email you in the next few weeks
  - We hope that you enjoy the workshop!

SpiNNaker Hardware & Software



Overview

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

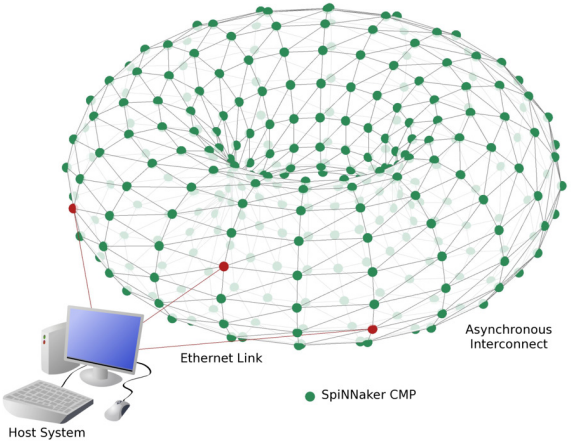
EPSRC

SpiNNaker

Contents

- What is SpiNNaker?
- SpiNNaker at different scales
- SpiNNaker architecture: chip & system
- Using SpiNNaker

SpiNNaker Project

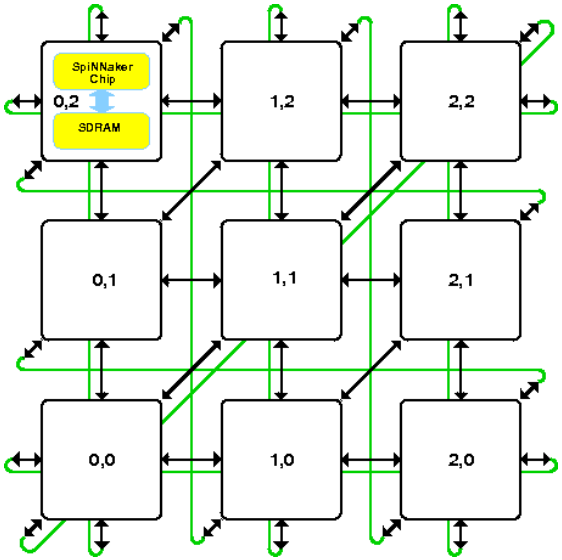


A million mobile phone processors in one computer
Able to model about 1% of the human brain…
…or 10 mice!

How is SpiNNaker Used?

- Some key user communities:
  - **Computational neuroscientists** to simulate large neural models and try to understand the brain
  - **Roboticists** to build advanced neural sensory and control systems
  - **Computer architects** to apply neural theories of computation to non-neural problems

SpiNNaker System



Chip-to-chip communications: Packet routing

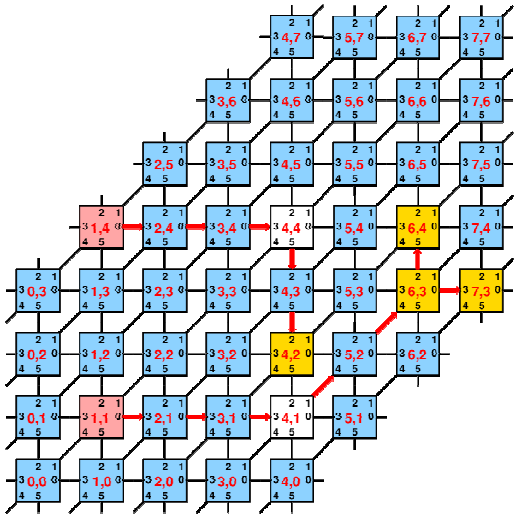
- No memory shared between chips!
- Communicate via simple messages called ***packets:***
  - *40 bit (no data) or*
  - *72 bit (includes 32-bit data word)*
- Four types of routing, most important (for you) is ***multicast***
- Packets used to communicate with the host and external peripherals:
  - Via Ethernet adapter for host comms.
  - Or via chip-to-chip SpiNNaker links for external devices

Routing Types

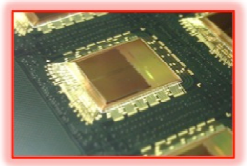
Nearest Neighbour
Point-to-Point
Multicast
Fixed Route

Multicast Routing

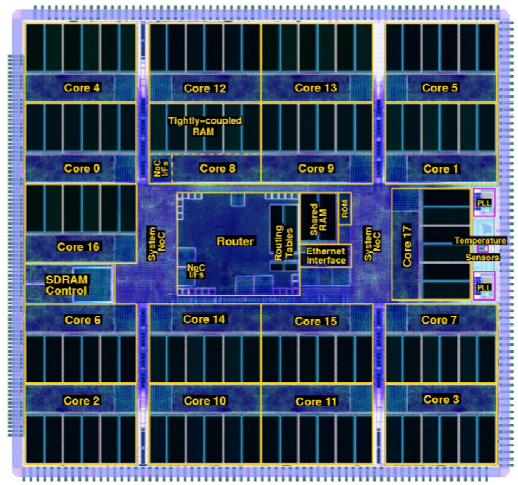
- Hardware router on each node
- Packets have a routing key
- Router has a look-up table of {*key, mask, data*} triplets
- If address matches a *key-mask* pair, the associated *data* tells router what to do with the packet



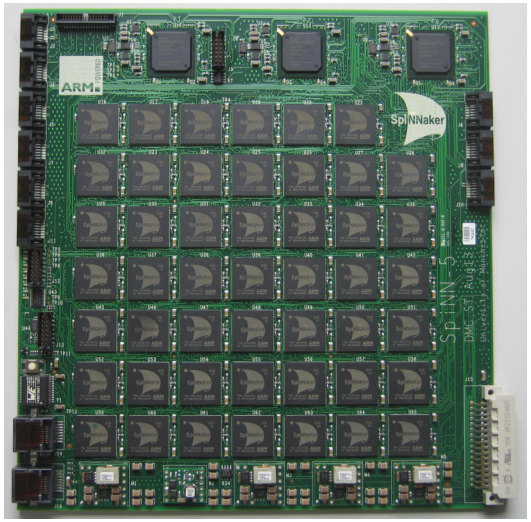
SpiNNaker Chip

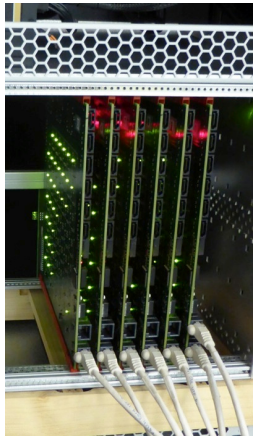


Multi-chip packaging by UNISEM Europe



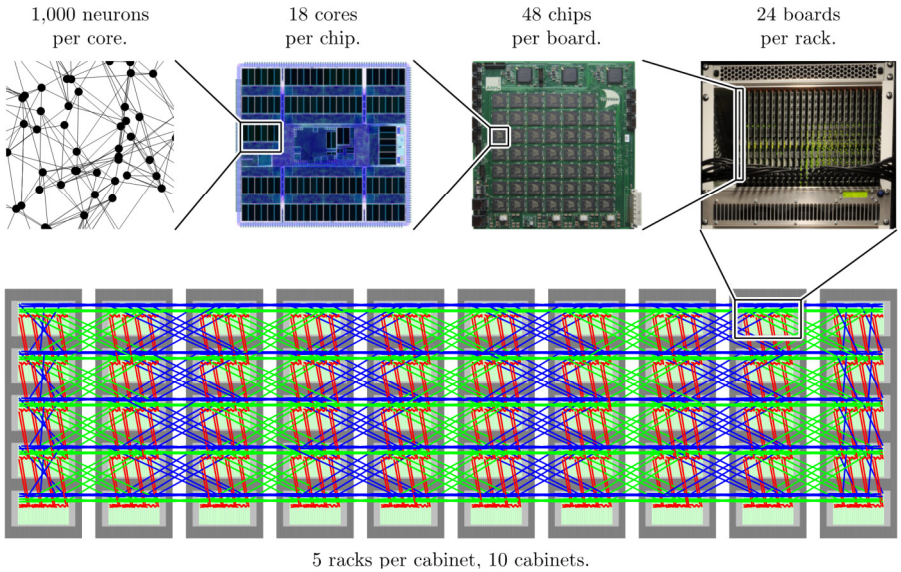
SpiNNaker Boards





SpiNNaker Machines

Scaling to a billion neurons



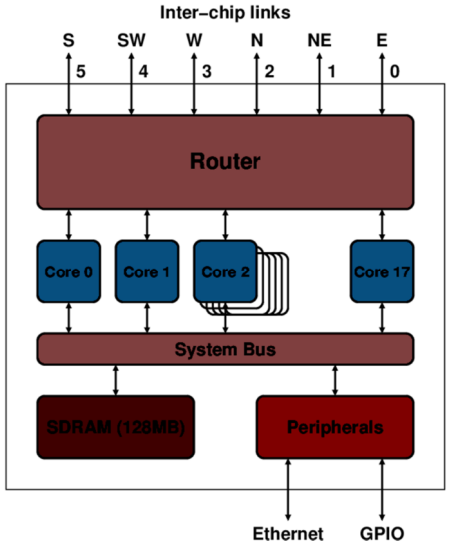
What Next for SpiNNaker?

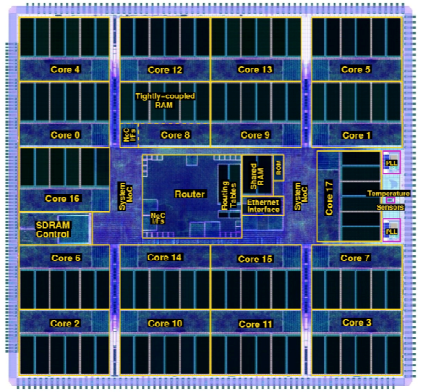
- Five cabinet machine (500K ARM cores)
  - Now online and available!
  - Open to any research project, in principle
- SpiNNaker2 being developed within HBP
  - New systems by 2020?
- For further information contact:

simon.davidson@manchester.ac.uk

Chip Architecture

SpiNNaker Node





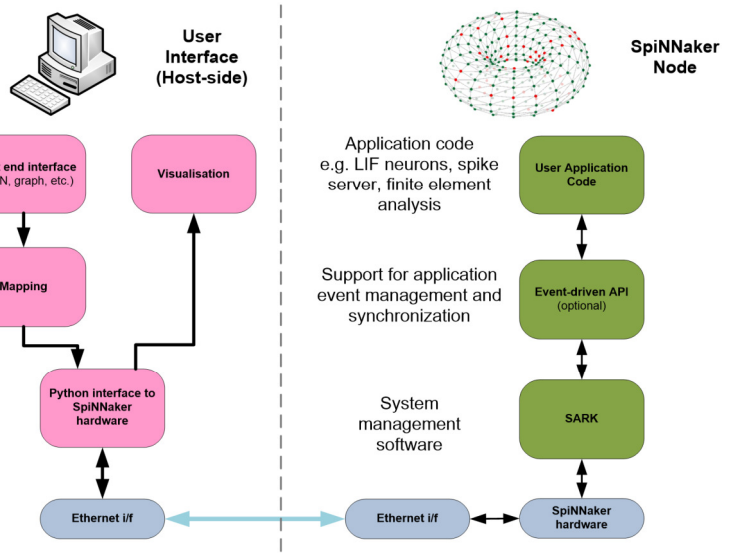
Chip Resources

- 18 cores on a chip:
  - 1 Monitor Processor
  - 16 Application processors
  - 1 fault-tolerant/yield spare
- Each core is an ARM968 processor
  - 200 MHz clock speed
  - No memory management or floating point!
  - Local memories:
    - 32K local code memory (ITCM), 64K local data (DTCM)
    - TCMs are visible only to local processor
- 128MByte SDRAM
  - Shared and visible to all processors on **same node**
- Router:
  - Directs flow of information from core-to-core across the machine

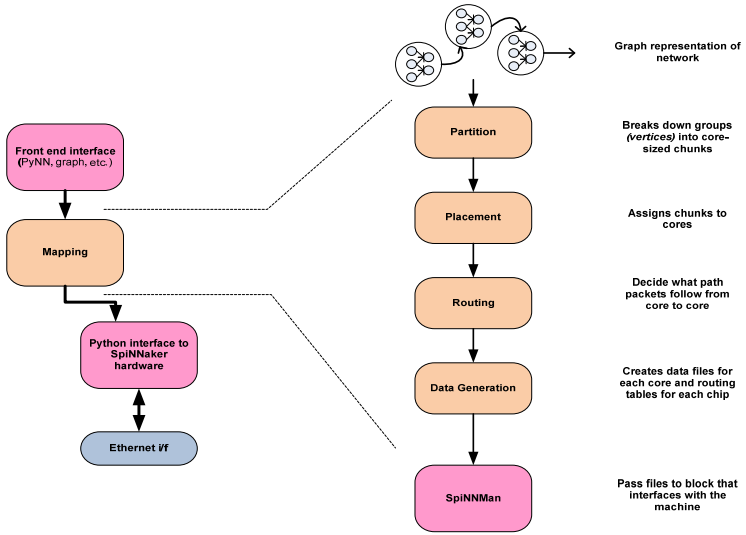
Using SpiNNaker:

The Software Stack

Software Stack



Mapping Process



What Files are Required for Simulation?

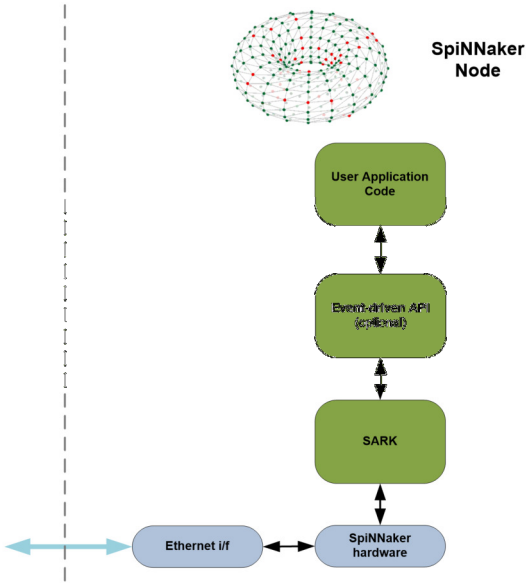
Files Required

For Each Core (16 per chip):

- Application C executable (32KB)
- Application local data (64KB)
- Application shared data (8MB)

For Each Chip:

- Router table
- Any shared data tables



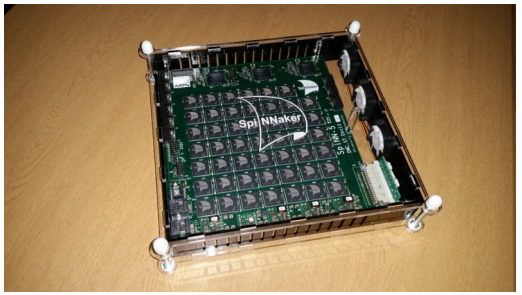
Order of Events (batch mode)

1. Compile network description
2. Map graph to machine
3. Generate data files
4. Load files
5. Synchronise the start on all cores!
6. Simulation runs to completion
7. Hands back control to host
8. Read back results and post-process

End of Overview!

- Much more detail on all of these topics
  - In the sessions to come….
- Any questions for now?
- Just one more thing to add….

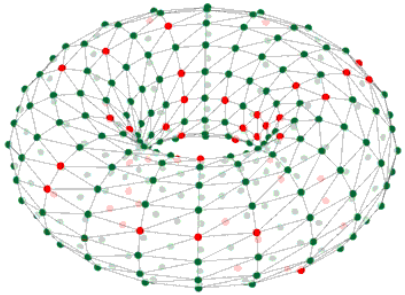
Buying SpiNNaker Hardware



- 48-node board now available for sale
- Non-commercial use only
- 4-node boards can only be loaned (currently!)

- For further information contact: simon.davidson@manchester.ac.uk

SpiNNaker Chip Resources



Steve Temple
SpiNNaker Workshop – Manchester – Sep 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

SpiNNaker

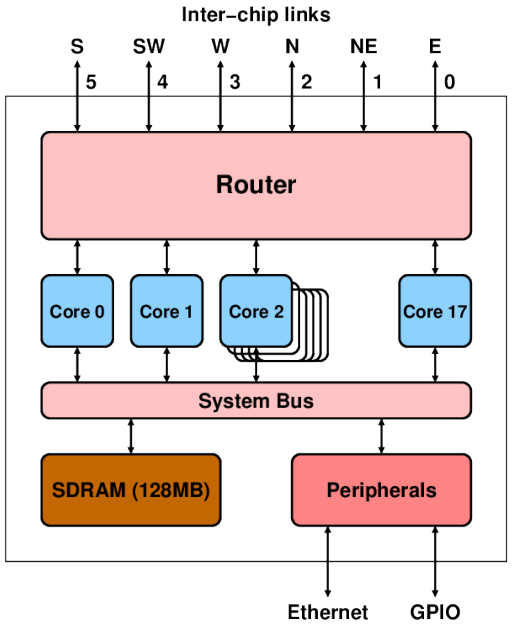
Overview

- Chip Architecture
- Core Architecture
- Low-level Communication
  - Packet formats
  - Multicast routing
- High-level Communication – SDP
- Hardware Limitations

Please interrupt if you have a question!

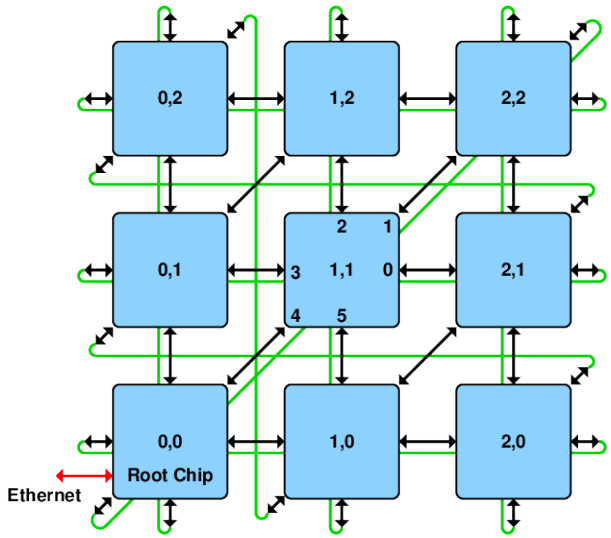
SpiNNaker

SpiNNaker Chip Outline



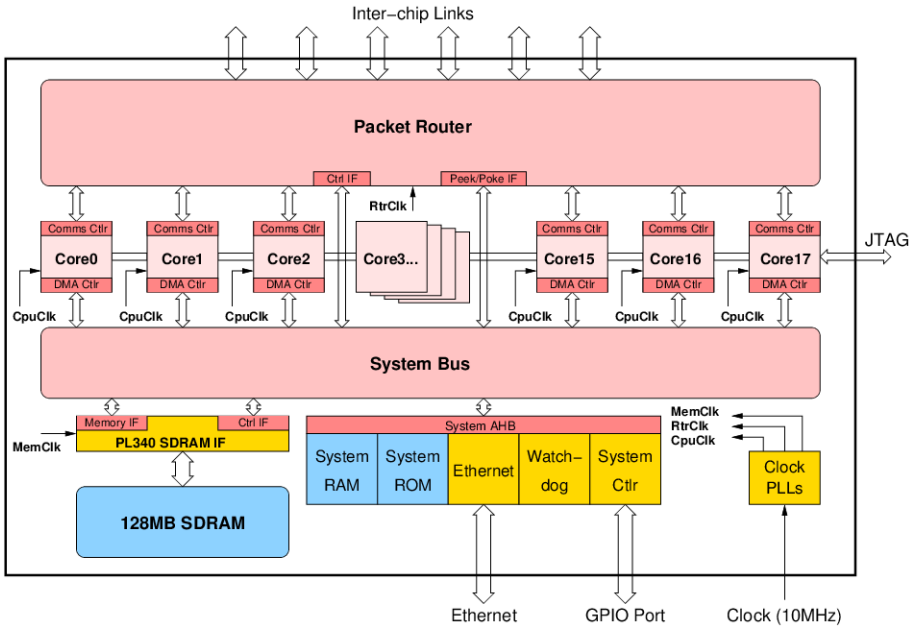
SpiNNaker

Chip Interconnect

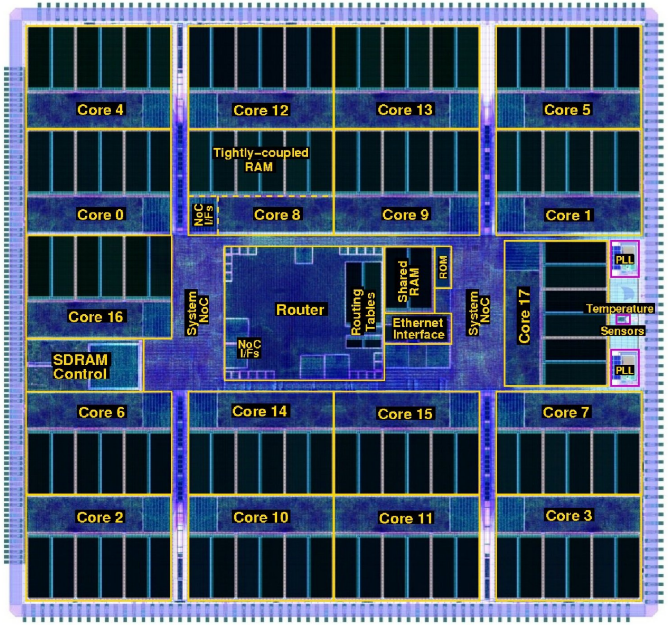


SpiNNaker

SpiNNaker Chip Details

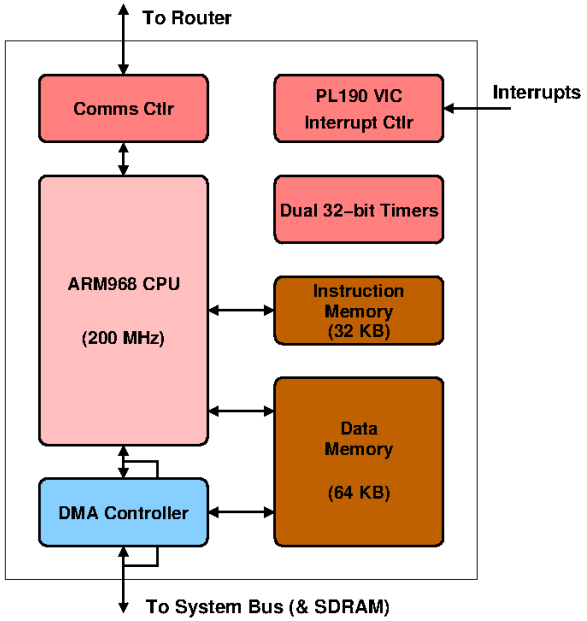


SpiNNaker Chip Layout



- 130nm process
- 10 x 10 mm
- 18 ARM cores with 96K SRAM
- Router
- SDRAM controller
- Asynchronous NoC

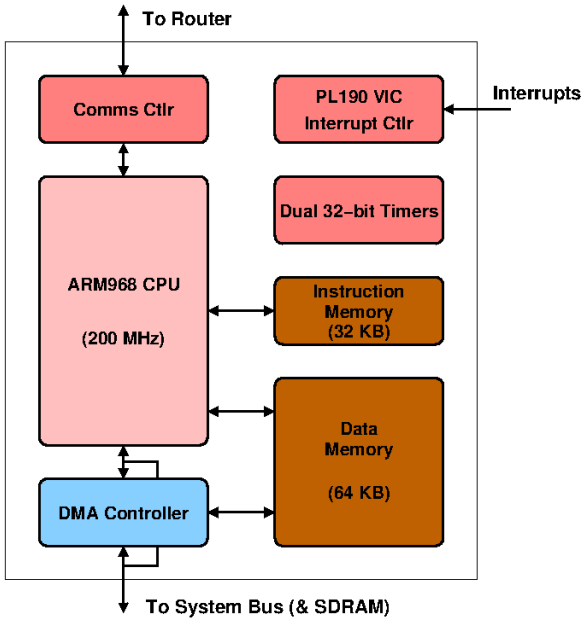
SpiNNaker Core



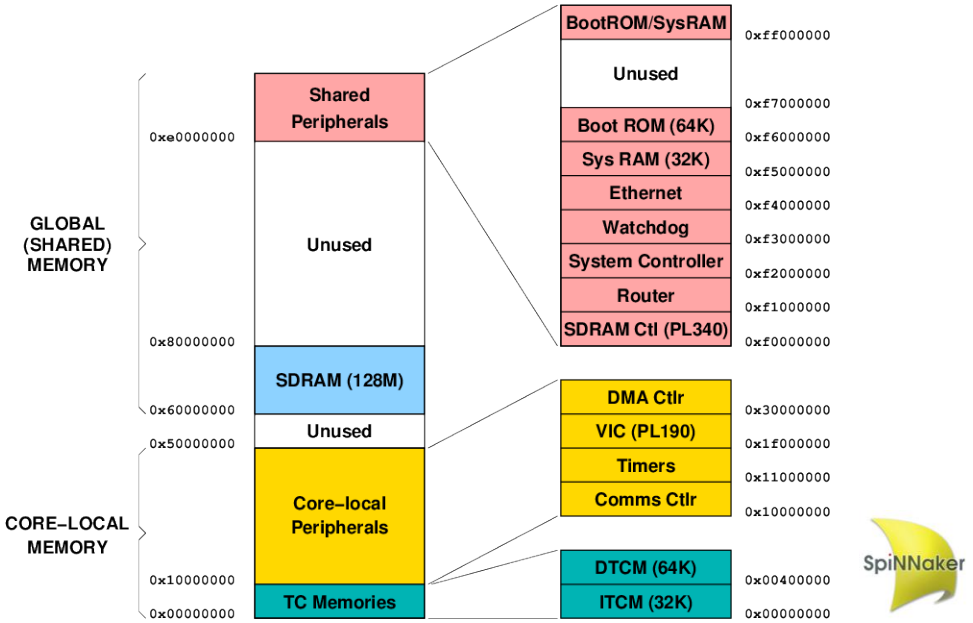
ARM968 CPU

- ARM9 CPU clocked at 200 MHz
- ARM v5TE architecture
  - Supports 32-bit ARM and 16-bit Thumb code
  - Some DSP instruction support - saturated arithmetic, extended multiplies
  - **No floating point hardware!**
- Two Tightly Coupled Memory (TCM) blocks
  - Single cycle (5 ns) access time
  - 32 KB Instruction TCM (ITCM)
  - 64 KB Data TCM (DTCM)
- DMA interface into both TCMs

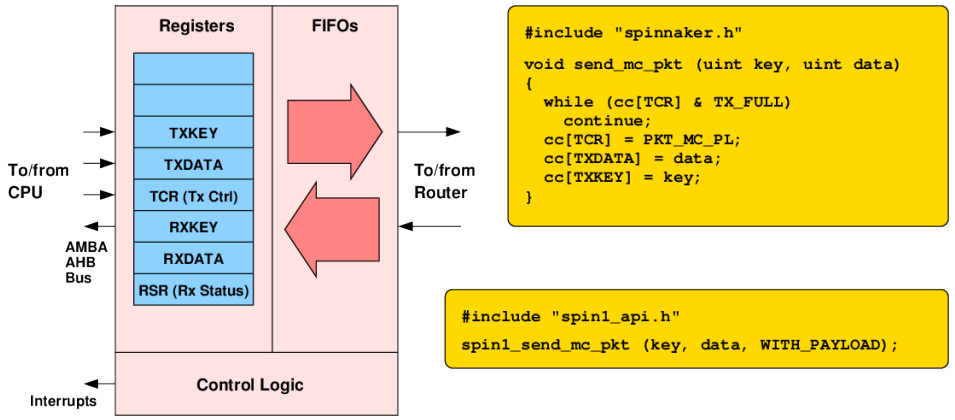
SpiNNaker Core



SpiNNaker Memory Map



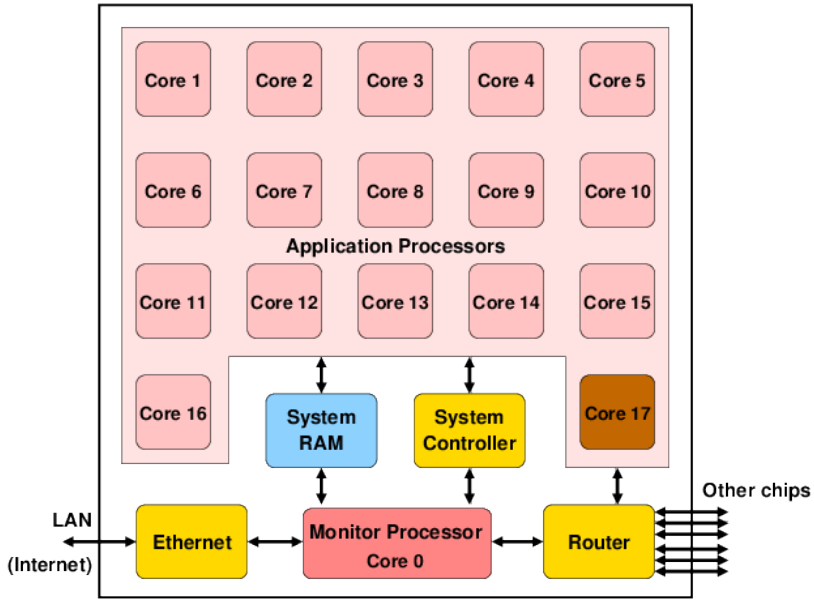
Communications Controller



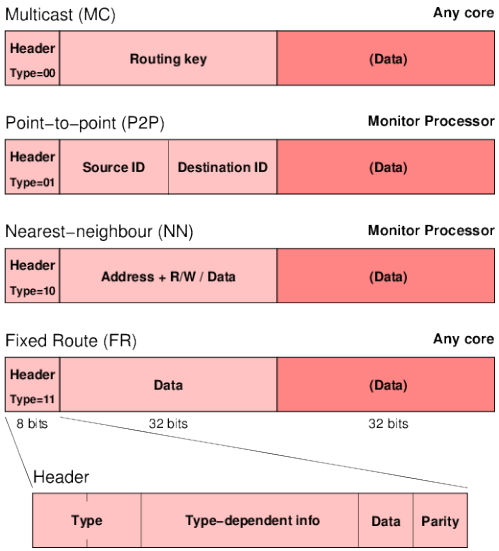

```
#include "spinnaker.h"

void send_mc_pkt (uint key, uint data)
{
  while (cc[TCR] & TX_FULL)
    continue;
  cc[TCR] = PKT_MC_PL;
  cc[TXDATA] = data;
  cc[TXKEY] = key;
}
```

```
#include "spin1_api.h"
spin1_send_mc_pkt (key, data, WITH_PAYLOAD);
```

Monitor Processor & Virtual Cores

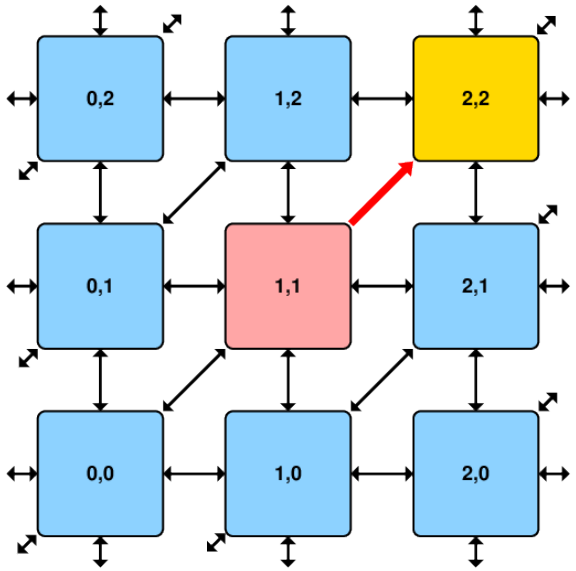


SpiNNaker Packet Types

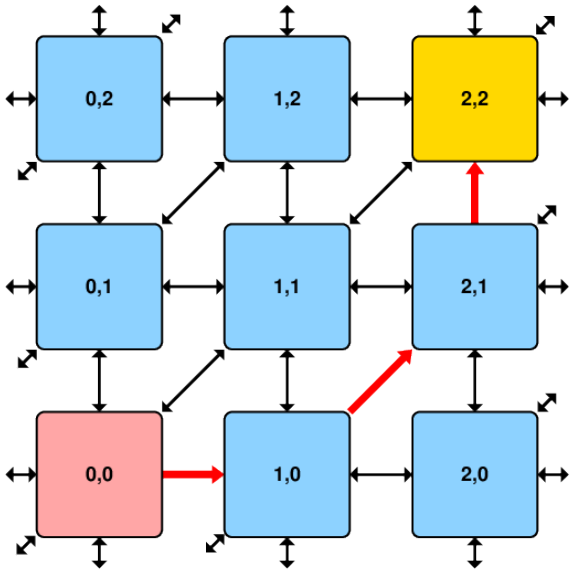



```
uint spin1_send_mc_pkt (uint key, uint data, uint payload);
```

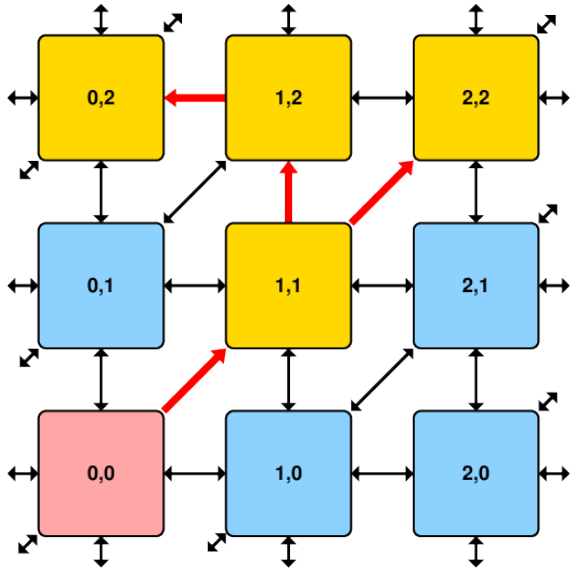
Nearest-neighbour packets



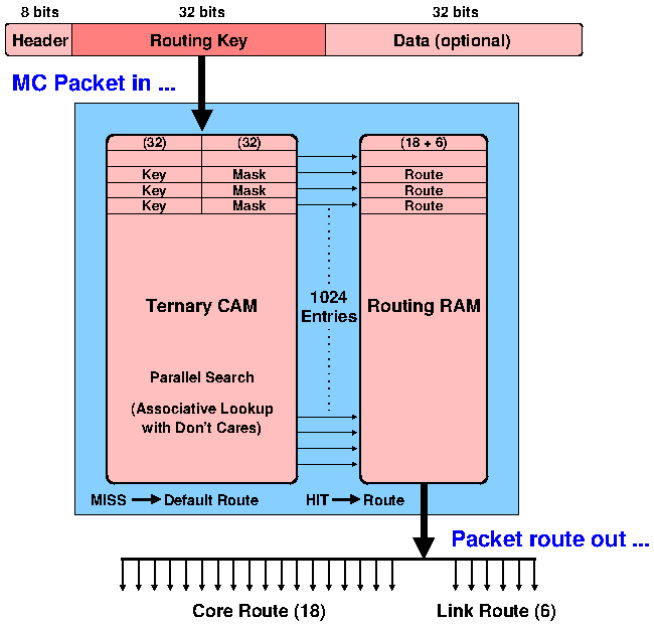
Point-to-point packets



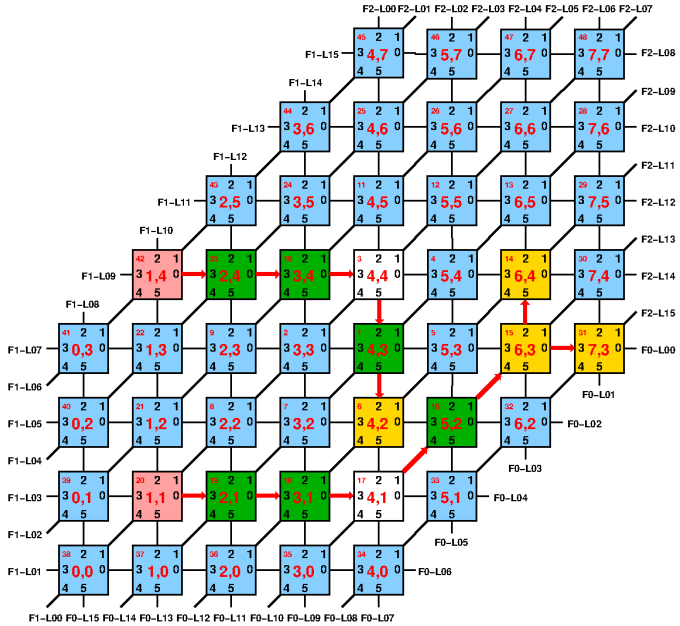
Multicast packets



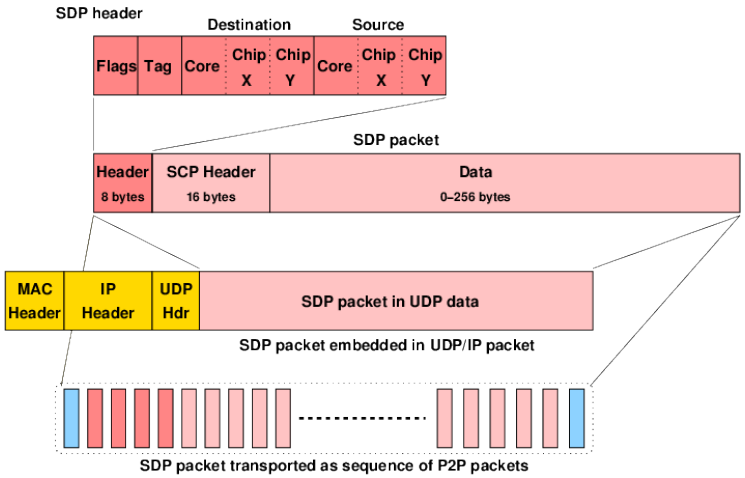
Multicast Packet Router



Multicast Packet Routing

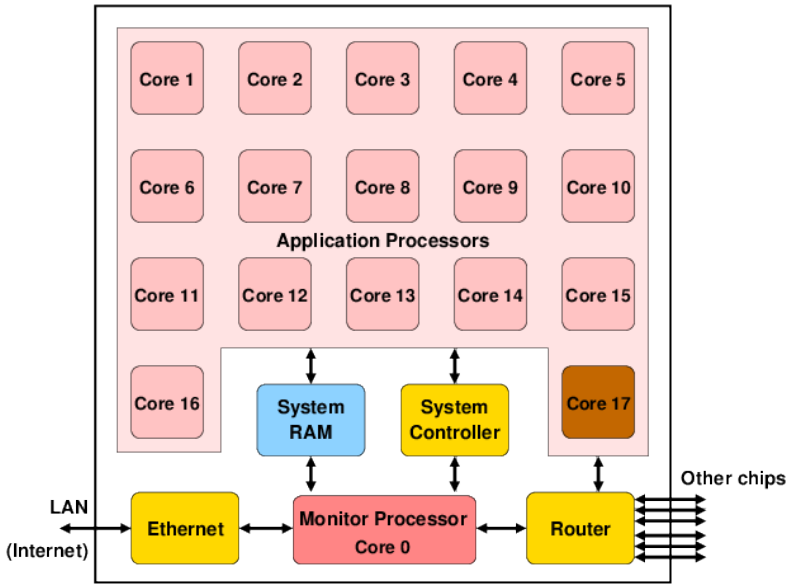


SpiNNaker Datagram Protocol




```
uint spin1_send_sdp_msg (sdp_msg_t *msg, uint timeout);
```

SDP Routing



SpiNNaker Hardware Limits

- Processors – 16/17 per chip (but scalable to thousands of chips)
- ARM968 – ARM9 at 200MHz – 220 DMIPS
- Local memory – very limited
  - Instruction memory – 32K bytes
  - Data memory – 64K bytes
- Local Memory access time - 5 ns
- Per chip memory – 128M bytes (shared)
- Shared memory access time
  - Individual accesses - > 100 ns (NB write buffer)
  - DMA accesses ~ 15ns per word

SpiNNaker Arithmetic Limits

- ARM968 has no floating point hardware
- Options
  - Soft Floating Point – slow and memory hungry
  - Fixed point – uses integer ops
    - Limited range before precision lost
    - Some GCC compiler support (but slowish)
    - Or hand code (C or assembly) for best performance (some libraries available)
- ARM968 has some DSP extensions
  - Saturation, MAC, double operations, CLZ
  - Accessible via compiler intrinsics

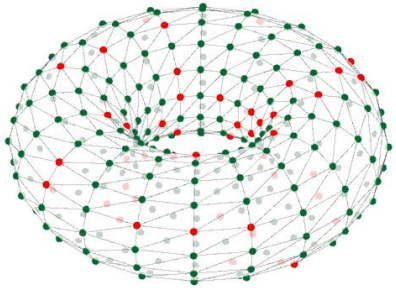
SpiNNaker Packet Limits

- Packet payload is small – typically 32 bits
- Packet bandwidth is limited
- Chip-to-chip links ~ 250M bit/s (5 or 3 M pkt/s)
  - Currently 50% slower via board-to-board links
- CPU packet processing overhead typically 200-1000ns
- Packets can get lost (dropped) in case of congestion – can be “re-injected” in some cases
- Multicast router table is not infinite!

SpiNNaker Bandwidth Limits

- Overall I/O bandwidth into the machine is limited
- Currently most external I/O is by 100 Mbit/s Ethernet (and only one interface per board)
- High level I/O via SDP is limited by software overheads
  - Around 10 Mbyte/s to Ethernet-attached chip
  - Around 2 Mbyte/s to 'unattached' chips (via P2P packets)
- Potential for higher I/O bandwidth via SATA links on FPGAs but currently unexploited

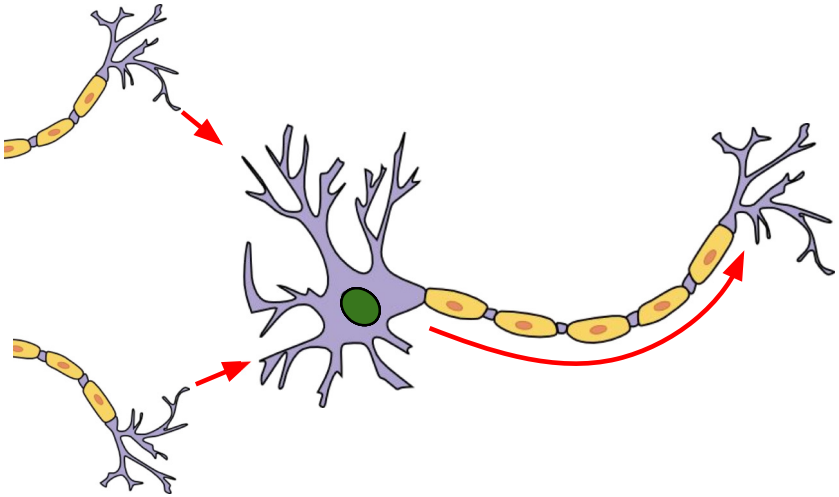
Running PyNN Simulations on SpiNNaker



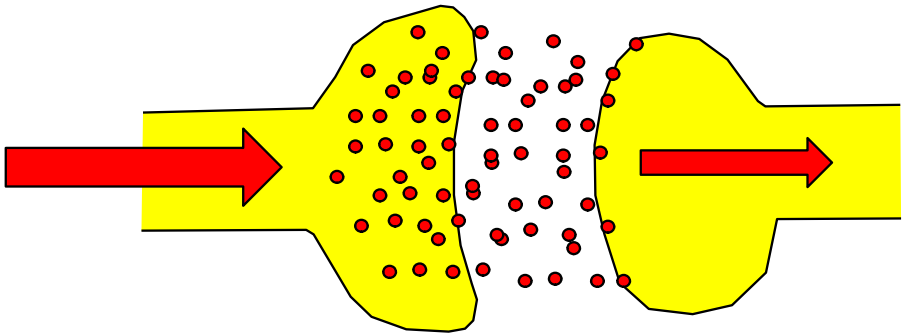
Andrew Rowley, Alan Stokes

SpiNNaker Workshop, September 2016

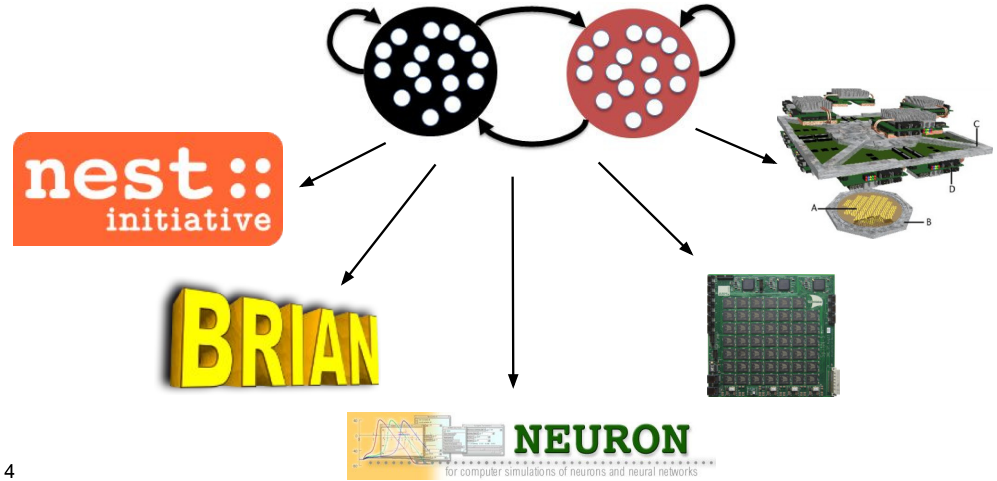
Spiking Neural Networks



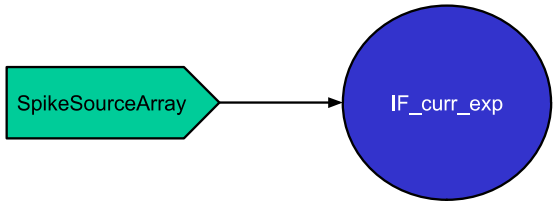
Spiking Neural Networks



What is PyNN?



A Simple PyNN Network



A Simple PyNN Network

```
import pyNN.spiNNaker as p
```


A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
```


A Simple PyNN Network

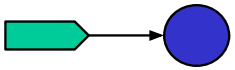
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
```

A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
```

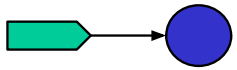
A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
```



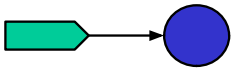
A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
```



A Simple PyNN Network

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
p.run(10)
```



Edit ~/.spynnaker.cfg

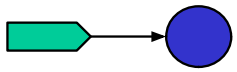
```
[Machine]
#-------
# Information about the target SpiNNaker board or machine:
# machineName:     The name or IP address of the target board
# version:         Version of the Spinnaker Hardware Board (1-5)
# machineTimeStep: Internal time step in simulations in usecs.
# timeScaleFactor: Change this to slow down the simulation time
#                      relative to real time.
#-------
machineName     = None
version         = None
#machineTimeStep = 1000
#timeScaleFactor = 1
```


Edit ~/.spynnaker.cfg

```
[Machine]
#-------
# Information about the target SpiNNaker board or machine:
# machineName:     The name or IP address of the target board
# version:         Version of the Spinnaker Hardware Board (1-5)
# machineTimeStep: Internal time step in simulations in usecs.
# timeScaleFactor: Change this to slow down the simulation time
#                      relative to real time.
#-------
machineName     = 192.168.240.253
version         = 3
#machineTimeStep = 1000
#timeScaleFactor = 1
```

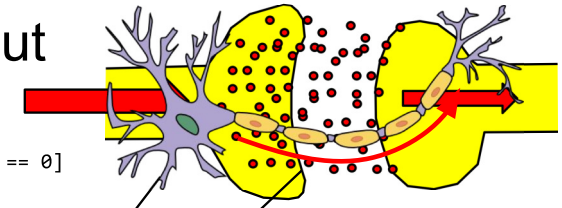

A Simple PyNN Network

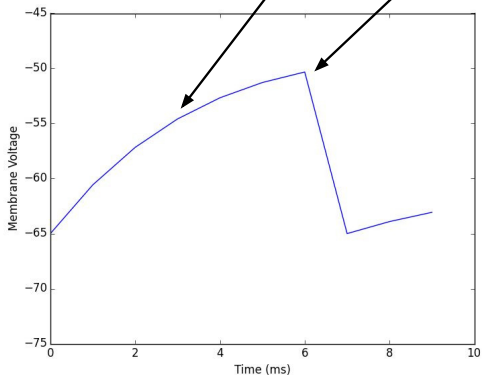
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1))
pop_1.record()
pop_1.record_v()
p.run(10)
spikes = pop_1.getSpikes()
v = pop_1.get_v()
```



Plotting Output

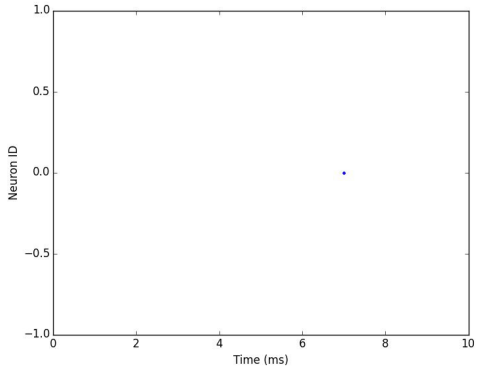
```
import pylab
time = [i[1] for i in v if i[0] == 0]
membrane_voltage = [i[2] for i in v if i[0] == 0]
pylab.plot(time, membrane_voltage)
pylab.xlabel("Time (ms)")
pylab.ylabel("Membrane Voltage")
pylab.axis([0, 10, -75, -45])
pylab.show()
```



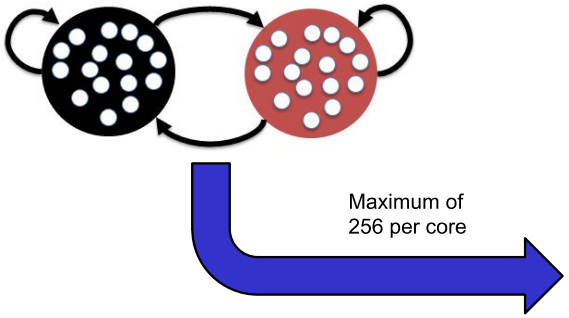


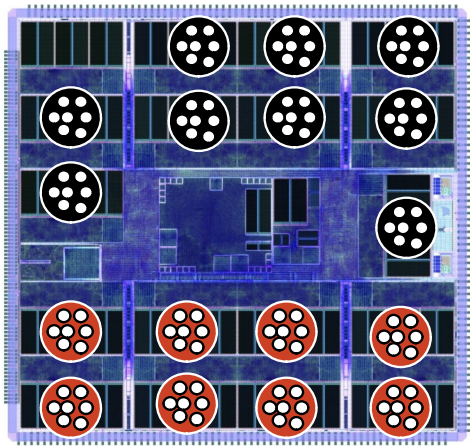
Plotting Output

```
import pylab
spike_time = [i[1] for i in spikes]
spike_id = [i[0] for i in spikes]
pylab.plot(spike_time, spike_id, ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 10, -1, 1])
pylab.show()
```

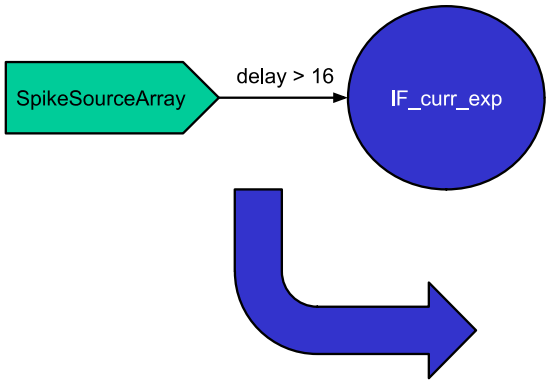


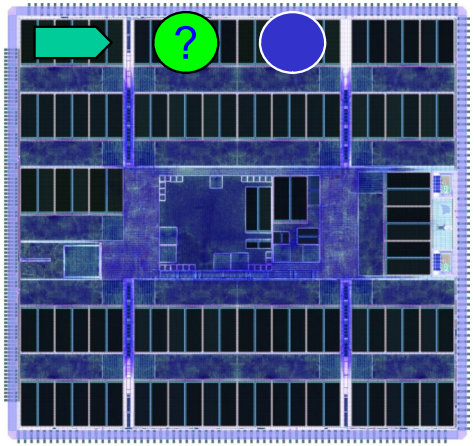
Limitations of PyNN on SpiNNaker:
Neurons Per Core





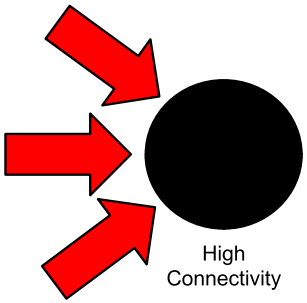
Limitations of PyNN on SpiNNaker:
Number of cores available

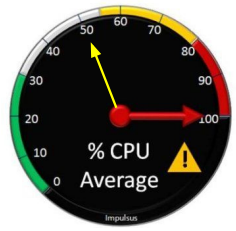




SpiNNaker-Specific PyNN

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p.set_number_of_neurons_per_core(p.IF_curr_exp, 100)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
```

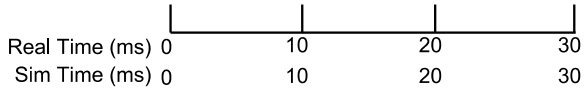




Plasticity

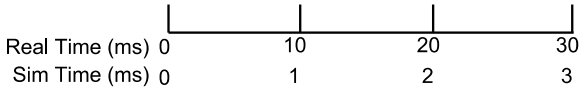
Configuration with spynnaker.cfg

```
[Machine]
machineName      = None
version          = None
timeScaleFactor  = 1
```

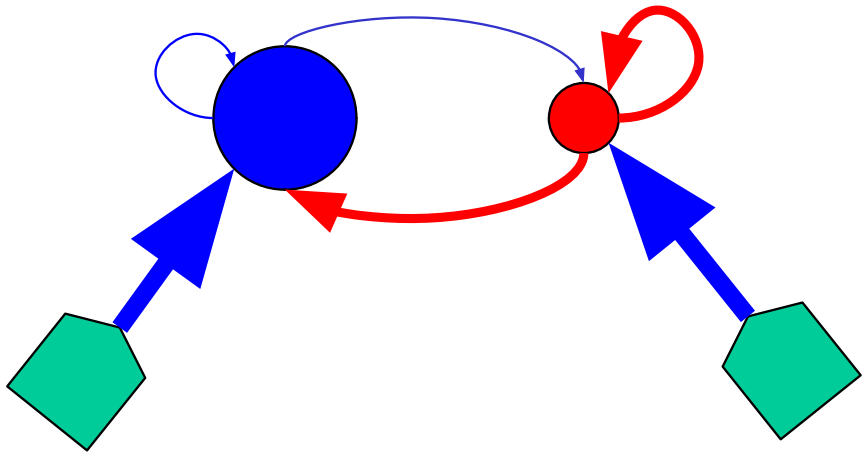


Configuration with spynnaker.cfg

```
[Machine]
machineName      = None
version          = None
timeScaleFactor  = 10
```



Balanced Random Network

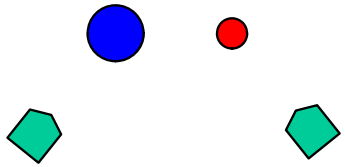


Balanced Random Network

```
import pyNN.spiNNaker as p
import pylab
from pyNN.random import RandomDistribution

p.setup(timestep=0.1)
n_neurons = 1000
n_exc = int(round(n_neurons * 0.8))
n_inh = int(round(n_neurons * 0.2))
```

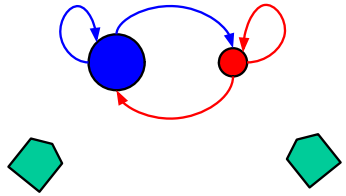
Balanced Random Network

```
pop_exc = p.Population(n_exc, p.IF_curr_exp, {},
                       label="Excitatory")
pop_inh = p.Population(n_inh, p.IF_curr_exp, {},
                       label="Inhibitory")
stim_exc = p.Population(n_exc, p.SpikeSourcePoisson,
                        {"rate": 10.0}, label="Stim_Exc")
stim_inh = p.Population(n_inh, p.SpikeSourcePoisson,
                        {"rate": 10.0}, label="Stim_Inh")
```



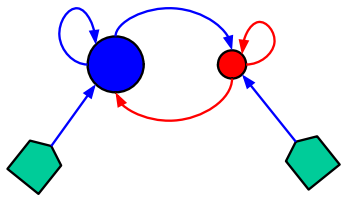
Balanced Random Network

```
conn_exc = p.FixedProbabilityConnector(0.1, weights=0.2,
                                       delays=2.0)
conn_inh = p.FixedProbabilityConnector(0.1, weights=-1.0,
                                       delays=2.0)
p.Projection(pop_exc, pop_exc, conn_exc, target="excitatory")
p.Projection(pop_exc, pop_inh, conn_exc, target="excitatory")
p.Projection(pop_inh, pop_inh, conn_inh, target="inhibitory")
p.Projection(pop_inh, pop_exc, conn_inh, target="inhibitory")
```



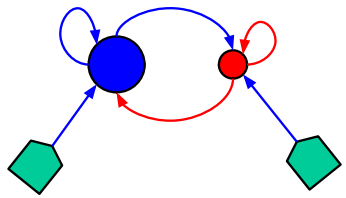
Balanced Random Network

```
delays_stim = RandomDistribution("uniform", [1.0, 1.6])
conn_stim = p.OneToOneConnector(weights=2.0,
                                delays=delays_stim)
p.Projection(stim_exc, pop_exc, conn_stim, target="excitatory")
p.Projection(stim_inh, pop_inh, conn_stim, target="excitatory")
```



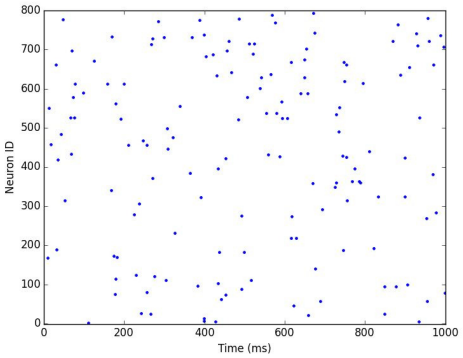
Balanced Random Network

```
pop_exc.initialize("v", RandomDistribution("uniform",
                                           [-65.0, -55.0]))
pop_inh.initialize("v", RandomDistribution("uniform",
                                           [-65.0, -55.0]))
pop_exc.record()
p.run(1000)
```

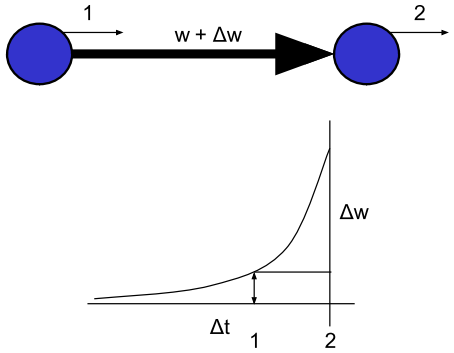


Balanced Random Network

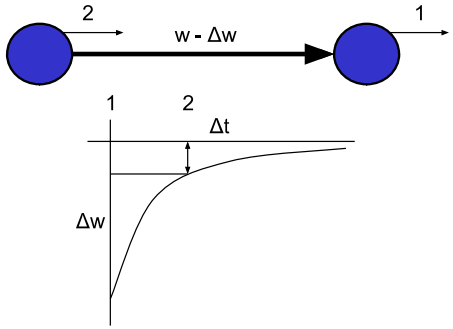
```
spikes = pop_exc.getSpikes()
pylab.plot([i[1] for i in spikes], [i[0] for i in spikes], ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 1000, -1, n_exc + 1])
pylab.show()
```



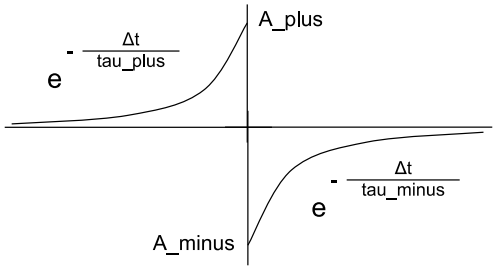
Spike Time Dependent Plasticity: Potentiation

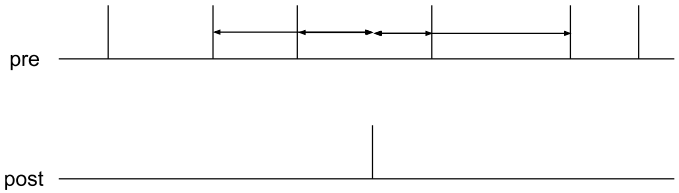


Spike Time Dependent Plasticity: Depression

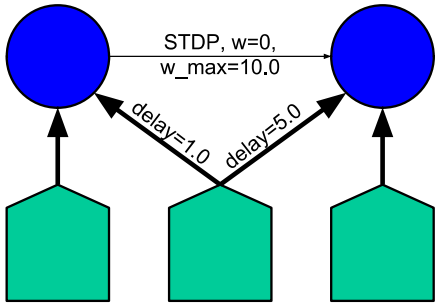


STDP Rules



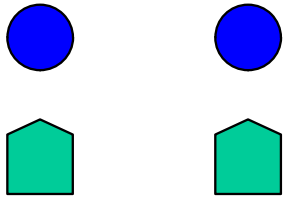


STDP in PyNN



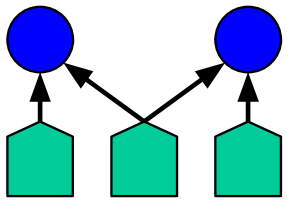
STDP in PyNN

```
import pyNN.spiNNaker as p
import pylab

p.setup(timestep=1.0)
n_neurons = 100

pre_pop    = p.Population(n_neurons, p.IF_curr_exp, {}, label="Pre")
post_pop   = p.Population(n_neurons, p.IF_curr_exp, {}, label="Post")
pre_noise  = p.Population(
    n_neurons, p.SpikeSourcePoisson, {"rate": 10.0}, label="Noise_Pre")
post_noise = p.Population(
    n_neurons, p.SpikeSourcePoisson, {"rate": 10.0}, label="Noise_Post")

pre_pop.record()
post_pop.record()
```



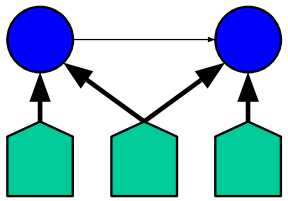
STDP in PyNN

```
training = p.Population(
    n_neurons, p.SpikeSourcePoisson,
    {"rate": 10.0, "start": 2000.0, "duration": 1000.0},
    label="Training")

p.Projection(pre_noise,  pre_pop,  p.OneToOneConnector(weights=2.0))
p.Projection(post_noise, post_pop, p.OneToOneConnector(weights=2.0))

p.Projection(training, pre_pop,  p.OneToOneConnector(weights=5.0, delays=1.0))
p.Projection(training, post_pop, p.OneToOneConnector(weights=5.0, delays=10.0))
```



STDP in PyNN

```
timing_rule = p.SpikePairRule(tau_plus=20.0, tau_minus=20.0)
weight_rule = p.AdditiveWeightDependence(
    w_max=5.0, w_min=0.0, A_plus=0.5, A_minus=0.5)

stdp_model = p.STDPMechanism(
    timing_dependence=timing_rule, weight_dependence=weight_rule)

stdp_projection = p.Projection(
    pre_pop, post_pop, p.OneToOneConnector(weights=0.0, delays=5.0),
    synapse_dynamics=p.SynapseDynamics(slow=stdp_model))
```



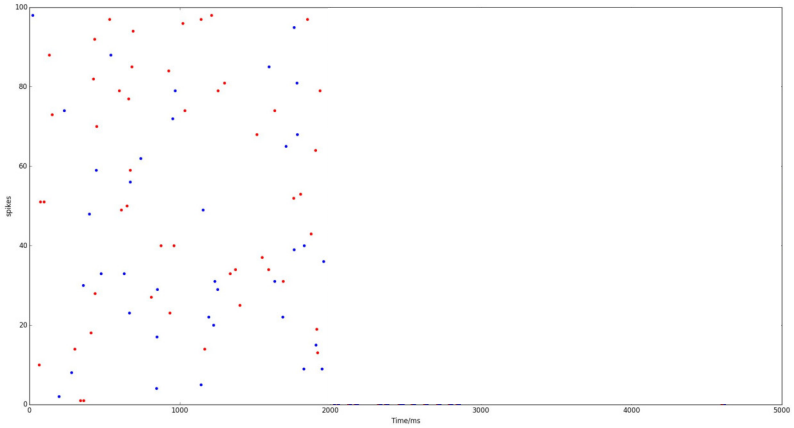
STDP in PyNN

```
p.run(5000)

pre_spikes = pre_pop.getSpikes()
post_spikes = post_pop.getSpikes()

print stdp_projection.getWeights()
```

```
[ 4.83886719  5.          4.27246094  5.          3.80957031  4.87060547
  5.          5.          5.          5.          5.
  4.25048828  5.          5.          4.57763672  5.          5.          5.
  5.          5.          5.          5.          5.          5.          5.
  5.          3.76953125  5.          5.          5.          5.          5.
  5.          5.          5.          5.          5.          5.          5.
  4.81591797  5.          5.          5.          5.          5.
  2.73339844  5.          5.          5.          5.          5.          5.
  5.          5.          5.          5.          5.          5.          5.
  5.          5.          5.          5.          4.98046875  5.          5.
  5.          5.          5.          5.          4.23388672  5.          5.
  5.          5.          5.          5.          4.69433594  5.          5.
  5.          5.          5.          5.          5.          5.          5.
  3.85400391  5.          5.          5.          4.07617188  5.          5.
  5.          5.          5.          5.          5.        ]
```


STDP in PyNN

```
pylab.figure()
pylab.xlim((0, 5000))
pylab.plot([i[1] for i in pre_spikes],  [i[0] for i in pre_spikes],  "r.")
pylab.plot([i[1] for i in post_spikes], [i[0] for i in post_spikes], "b.")
pylab.xlabel('Time/ms')
pylab.ylabel('spikes')
pylab.show()
```



6th SpiNNaker Workshop

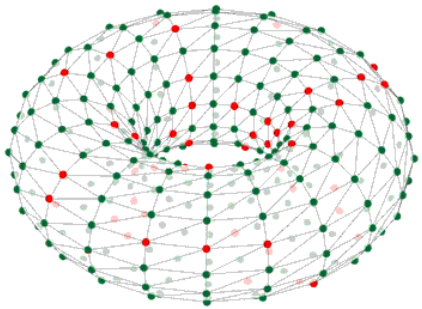
Day 2

September 6th 2016

	Session	Presenter
09:00	SpiNNaker system software (SARK)	ST
10:00	Coffee (earlier than usual)	
10:30	SpiNNaker API + event driven simulation	LAP
11:30	Writing Applications on SpiNNaker - Overview	SD
12:00	Lunch	
13:00	Introduction to Graph Front End (GFE)	ABS (AGR)
14:00	ybug and gdb walk-through	ST
15:00	Coffee	
15:30	Lab time	
16:30	Close	

Manchester, UK

SpiNNaker System Software



Steve Temple
SpiNNaker Workshop – Manchester – Sep 2016

European Research Council
Established by the European Commission
Human Brain Project
EPSRC

Overview

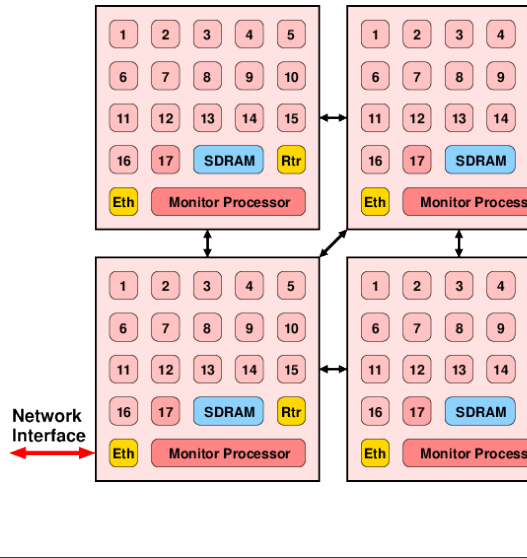
- SpiNNaker applications and their environment
- SC&MP, *ybug* and application loading
- SARK (SpiNNaker Application Runtime Kernel)
  - Application start-up
  - SARK function library
  - Examples
  - Documentation

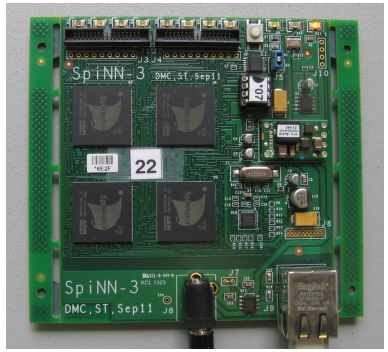
Please interrupt if you have a question!

Building Applications

- Languages – mostly C with bits of assembler
- Toolchain choice
  - ARM tools – RVDS 4 and DS-5 (free for academics)
  - GCC – GNU ARM Embedded Toochain (free)
- Library support
  - Toolchain libraries – C library functions, maths, etc
  - SARK – low-level SpiNNaker support library
  - Spin1 API – event-based application library
- Linking – support libs + application code
  - Creates application to be loaded
  - Application file format is APLX

Execution Environment (1)

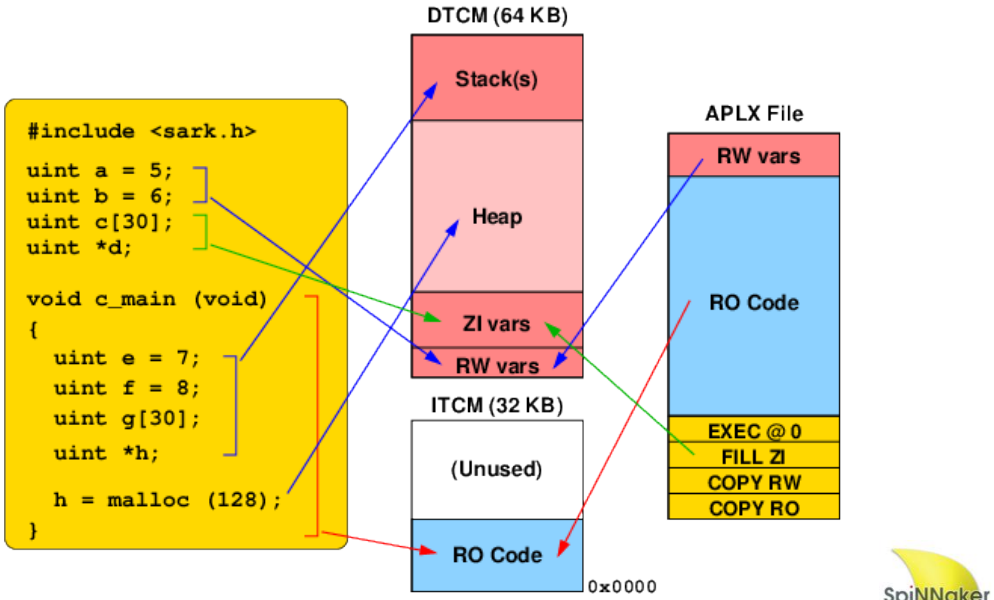




Execution Environment (2)

- One application per core
- Executable code (instructions) in ITCM (32 KB)
- Data (variables, stacks, heap) in DTCM (64 KB)
- Bulk and/or shared data in SDRAM (128 MB)
- Code/data access from ITCM/DTCM is fast (5 ns)
- Data access to SDRAM is slow (> 100 ns) and subject to contention
- DMA controller in each core can move bulk data between I/DTCM and SDRAM faster (~ 15 ns/word) without requiring CPU

Mapping Program to Memory



SC&MP

- "SpiNNaker Control & Monitor Program"
- Loaded onto all Monitor Processors during bootstrap
- Communicates with host computer using SCP (SpiNNaker Command Protocol) over SDP
- Supervises operation of a single chip
- Allows program loading to Application Cores
- Acts as router for SDP packets between any pair of cores or with external Internet endpoints
- Flashes the LED!

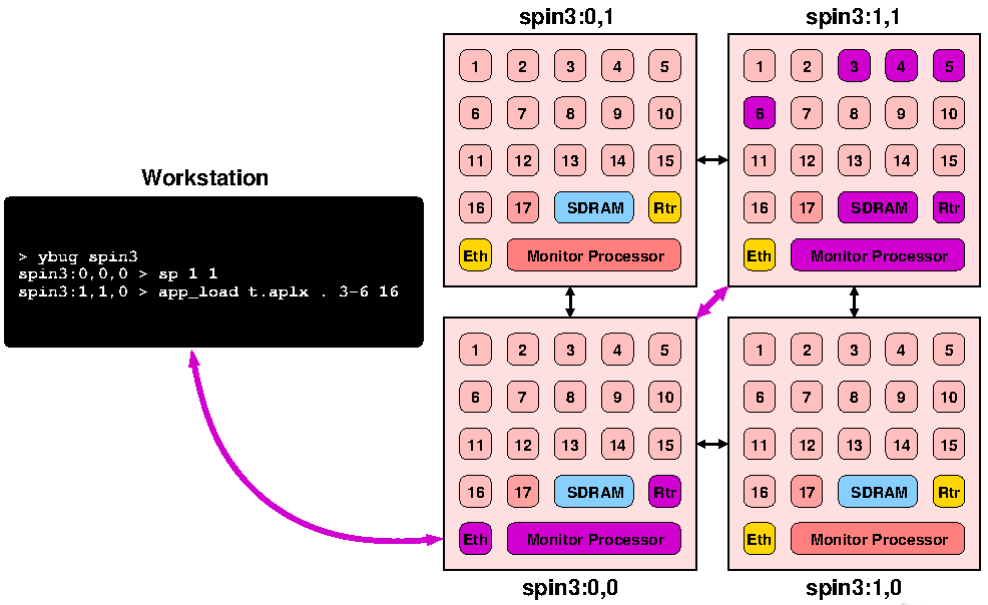
SC&MP, SCP and *ybug*

- SC&MP provides command interface via SCP
  - `Ver` – give S/W version, etc
  - `Read (addr, length)` – read SpiNNaker memory
  - `Write (addr, length, data)` – write SpiNNaker memory
  - `Reset (core_mask)` – reset Application Cores
- Host (workstation) embeds SCP/SDP in UDP/IP to talk to SpiNNaker Monitor Processor on the Root Chip
- *ybug* is a simple command-line tool which runs on a workstation and provides an interface to SC&MP for application loading and debug

Application Loading (1)

- *ybug* sends the application APLX to the relevant SpiNNaker chips.
- The APLX image is copied to a known place in shared memory
- *ybug* requests that the relevant Application cores are reset.
- The reset code is an *APLX loader* which loads the image according to instructions in the APLX header
- This usually results in the application being copied into ITCM and entered at address zero (the ARM reset vector)

Application Loading (2)



SARK

- SpiNNaker Application Runtime Kernel
- Three main functions
  1) Application start-up
  2) Library of useful functions
  3) Communication via SDP with Monitor Processor (and hence rest of system)
- SARK is automatically linked with applications when they are built
- Occupies around 3 KB in the image

Application Start-Up

- Start-up code at start of ITCM is SARK
  - Configures stacks for 4 ARM execution modes
  - Initialises Heap and SDP message buffers in DTCM
  - Initialises shared-memory data structure (VCPU)
  - Calls a function to do pre-application set-up
  - Calls the function **`c_main`**, the application entry point
  - Calls a function to do post-application clean-up
  - Goes to sleep!
- Some applications will never terminate
- SARK provides SDP communications with the application

SARK Library (1)

- CPU control
  - Interrupt disabling and enabling
  - Entering low power (sleep) mode
- Memory manipulation
  - Memory copy and fill (small footprint)
  - SDP message copying
- Pseudo-Random number generation (32-bit)
- SDP messaging
  - Message allocation in DTCM and shared memory
  - SDP message transmission

SARK Library (2)

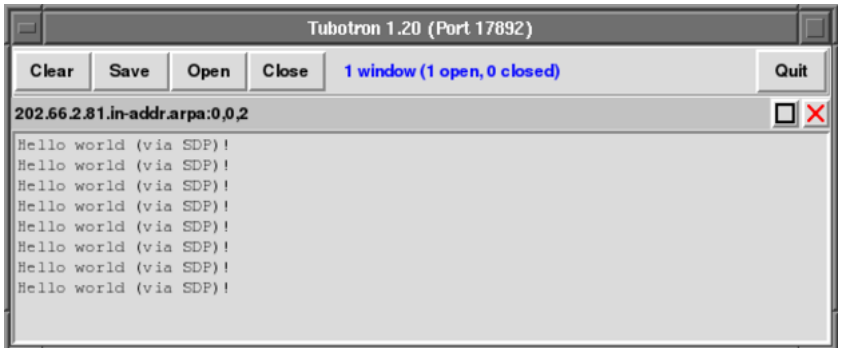
- Text output via “printf”
  - Text sent to a host system using SDP packets
  - Text buffered in SDRAM
- Hardware locks and semaphores
- Memory management
  - `malloc/free` for DTCM heap
  - `malloc/free` for shared memories (eg SDRAM) with locking
  - `malloc/free` for router MC routing table
- Environment queries
  - What is my core ID, chipID, etc

SARK Library (3)

- Hardware interfaces
  - LED control
  - Router control – setting MC and P2P table entries
  - VIC control – allocating interrupt handlers to specific hardware interrupts
- Timer management
  - Routines to schedule/cancel events at some time in the future
- Event management
  - Routines to associate events with interrupts
  - Management of priority event queues

SARK – Example 1

```
#include <sark.h>

void c_main (void)
{
  io_printf (IO_STD, "Hello world (via SDP)!\n");
  io_printf (IO_BUF, "Hello world (via SDRAM)!\n");
}
```



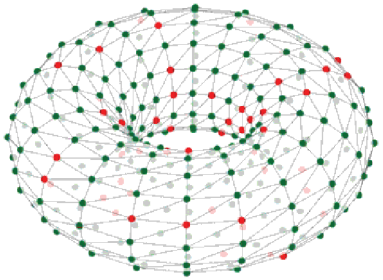
SARK - Example 2

```
#include <sark.h>

INT_HANDLER timer_int_han (void)
{
  tc[T1_INT_CLR] = (uint) tc;      // Clear interrupt in timer
  sark_led_set (LED_FLIP (1));     // Flip a LED
  vic[VIC_VADDR] = (uint) vic;     // Tell VIC we're done
}

void timer_setup (uint period)
{
  tc[T1_CONTROL] = 0xe2;                // Set up count-down mode
  tc[T1_LOAD] = sark.cpu_clk * period;  // Load time (us)
  sark_vic_set (SLOT_0, TIMER1_INT, 1, timer_int_han);
}

void c_main ()
{
  io_printf (IO_STD, "Timer interrupt example\n");
  timer_setup (500000);       // (0.5 secs)
  cpu_sleep ();               // Send core to sleep
}
```

Documentation & Help

- SARK – notes in SpiNNaker Tools - `docs/sark.pdf`
- *ybug* – user guide in SpiNNaker Tools – `docs/ybug.pdf`
- `"spinnaker.h"` - describes the SpiNNaker hardware – memory maps, peripheral registers...
- `"sark.h"` describes all SARK data structures and functions. Commented in Doxygen style.
- All source code is provided...
- If desperate, talk to us!

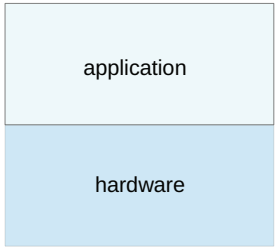
SpiNNaker API and event-driven simulation

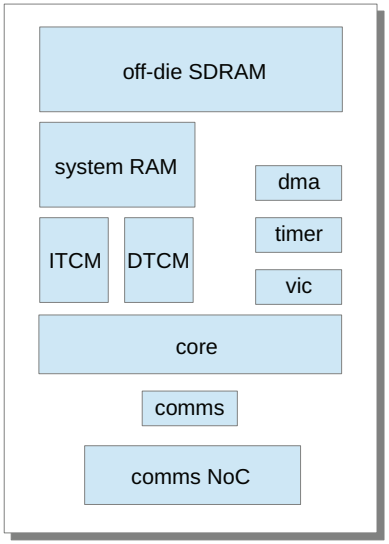


Luis Plana

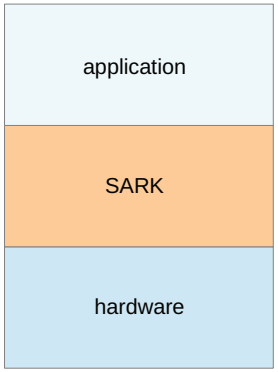
SpiNNaker Workshop, September 2016

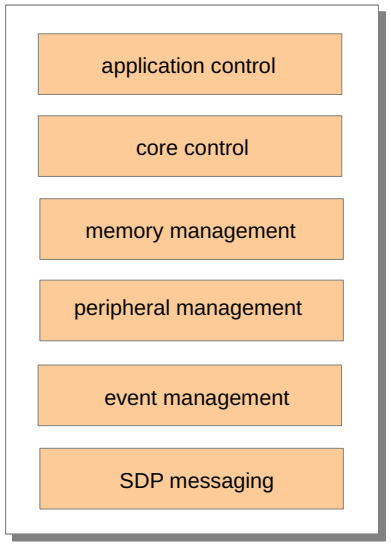
hardware resources



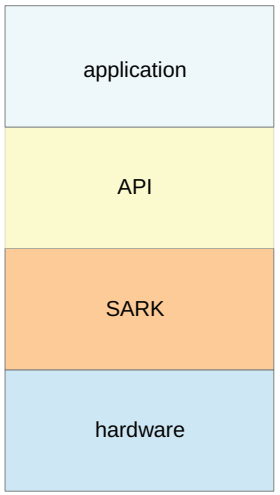


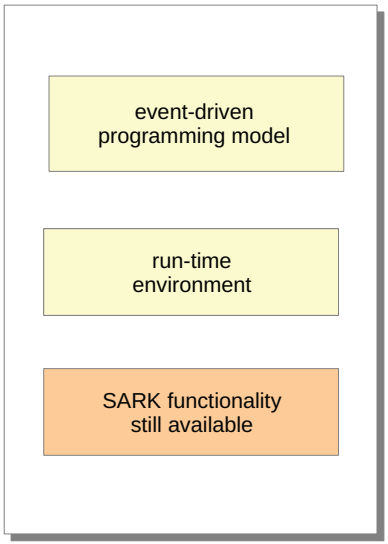
SARK: low-level software





API: run-time environment





event-driven model

applications do not control execution flow

applications indicate functions to be executed when events of interest occur

API controls execution and schedules application functions when appropriate

application functions are known as callbacks

events and callbacks

event	trigger
timer tick	periodic event has occurred
multicast packet received	multicast packet has arrived
DMA transfer done	scheduled DMA transfer completed successfully
SDP packet received	SDP packet has arrived
user event	application-triggered event has occurred

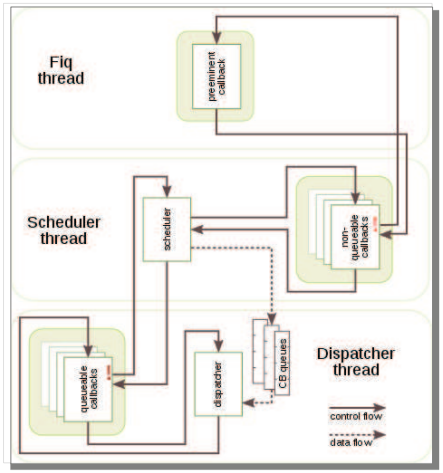
event	first argument	second argument
timer tick	simulation time (ticks)	null
MCP w/o payload received	key	0
MCP with payload received	key	payload
DMA transfer done	transfer ID	tag
SDP packet received	*mailbox	destination port
user event	arg0	arg1

priorities

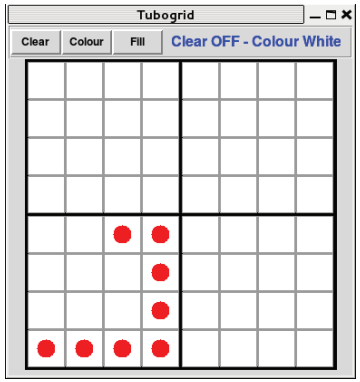
priority level = -1
only one callback
cannot be pre-empted

priority level = 0
can only be pre-empted
by priority -1 callback

priority level > 0
can be pre-empted
by priority <= 0 callbacks
scheduled in priority order



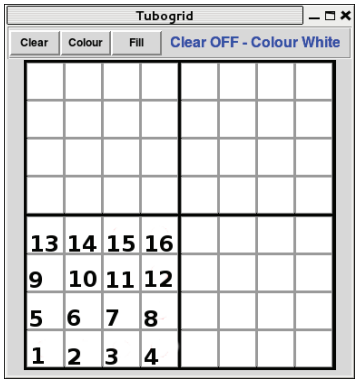
first program



first

```
// circle sequence
uint circle_pos[] =
{
  1, 2, 3, 4, 8, 12, 16, 15,
  14, 13, 9, 5, 6, 7, 11, 10
};

// iterate over 16 positions
for (uint i = 0; i < 16; i++)
{
  // update display,
  print_circle (circle_pos[i]);

  // and delay next circle
  for (uint j = 0; j < BIG_NUM; j++)
  {
    continue;
  }
}
```

distributed program



each core

```
// circle sequence
uint circle_pos[] =
{
  1, 2, 3, 4, 8, 12, 16, 15,
  14, 13, 9, 5, 6, 7, 11, 10
};

// this core's id
id = spin1_get_core_id();

// delay my circle,
for (uint j = 0; j < (id * BIG_NUM); j++)
{
  continue;
}

// and update display
print_circle (circle_pos[id]);
```

event-driven program

c_main

```
// 0.125s tick period (in microseconds)
#define TIMER_TICK_PERIOD  125000

void c_main()
{
  // initialize variables and state
  // ---------------------------------------
  id = spin1_get_core_id();
  my_state = OFF;
  old_state = my_state;

  // prepare for execution
  // ---------------------------------------
  // set timer tick value
  spin1_set_timer_tick (TIMER_TICK_PERIOD);

  // register callbacks
  spin1_callback_on (
    MC_PACKET_RECEIVED, packet, -1);

  spin1_callback_on (
    TIMER_TICK, timer, 0);

  // go
  // ---------------------------------------
  spin1_start(SYNC_WAIT);
}
```

packet callback

```
void packet (uint pkt_key, uint pkt_payload)
{
  // update my state
  my_state = ON;
}
```

timer callback

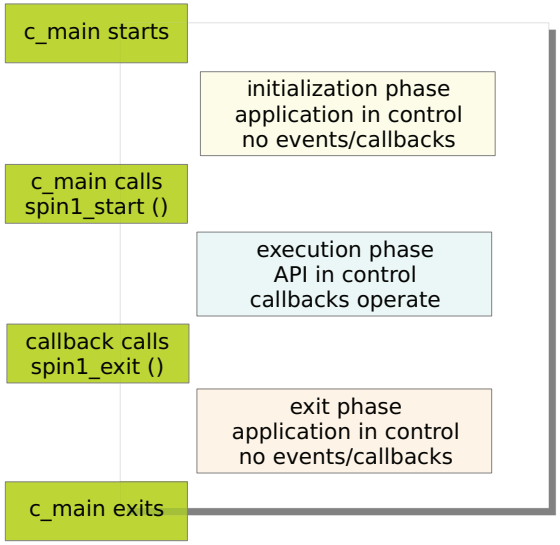
```
void timer (uint ticks, uint b)
{
  // check if state changed
  if (my_state != old_state)
  {
    // update display,
    print_circle (circle_pos[id]);

    // send a packet to next core in the chain,
    spin1_send_mc_packet(my_key, 0, NO_PAYLOAD);

    // and remember state
    old_state = my_state;
  }
}
```

additional support

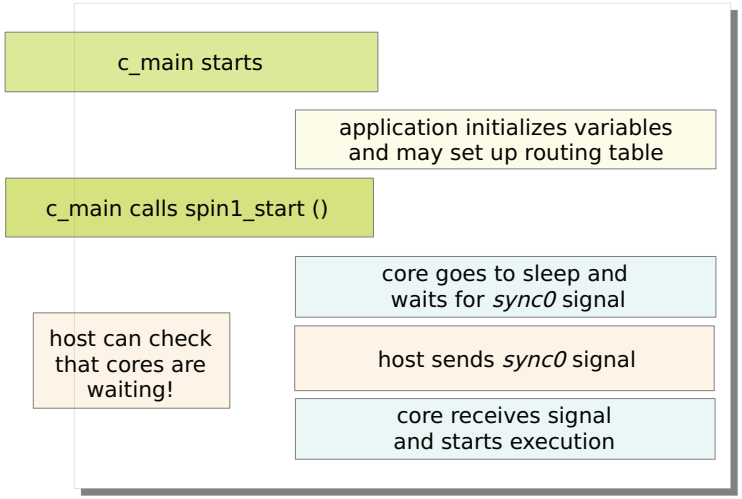
function	use
start/stop execution	start and stop simulation
set timer period	real-time or periodic callback
send multicast packet	inter-core communications
send SDP packet	host or I/O peripheral communications
start DMA transfer	software-managed cache
trigger user event	start a callback with priority <= 0
schedule callback	start a callback with priority > 0
enable/disable interrupts	critical section access (inter-thread control)
provide chip address and core ID	find out who you are
configure multicast routing table	setup routing entries

see API documentation for complete list

program structure



synchronization barrier



to think about: pitfalls

asynchronous operation and communications

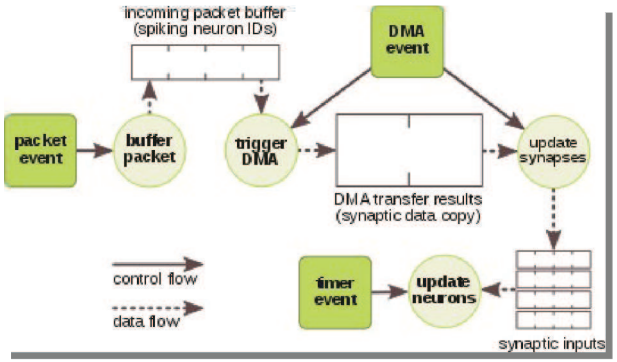
multicast packets can be dropped due to congestion

UDP-based I/O *not guaranteed!*

no floating-point support use fixed-point arithmetic

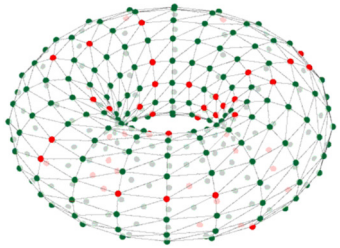
no globally-shared resources use message passing

example: spiking neural network



what is a sensible choice of priorities?

Writing an Application for SpiNNaker - Introduction



Simon Davidson, Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

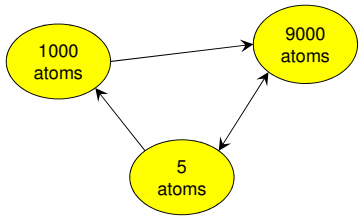
EPSRC

SpiNNaker

Contents

View of an application distributed across parallel processors

- Two main activities:
  - Computation
  - Communication
- Think of the problem as a graph:
  - Vertex = computation node
  - Edge = flow of information between nodes
- Node can hold a collections of objects of the same type, which we call ***atoms***
- e.g. Many spiking neurons in one population



Design Process for New Applications

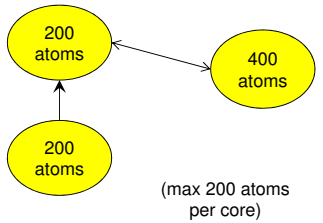
- **The application designer** creates components (nodes and communication types)
- These components plug into our tool chain
- A **user** can then invoke the Graph Front End (GFE) to create and run their own networks on SpiNNaker
- Input is textual, like a PyNN script, in which the user instantiates the components created by the application designer
- Graph Front End is NOT a Graphical Interface - No GUI!

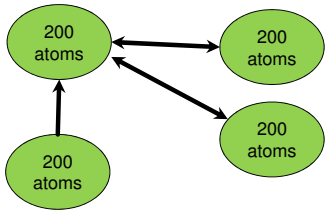
Example script: Conway's Game of Life

```
import spinnaker_graph_front_end as front_end
import sys

# set up the front end and ask for a machine with 48 chips
front_end.setup()

cell_1 = MyCell()
cell_2 = MyCell()
edge = MachineEdge(cell_1, cell_2)
front_end.add_machine_vertex_instance(cell_1)
front_end.add_machine_vertex_instance(cell_2)
front_end.add_machine_edge_instance(edge, "STATE")

# run the simulation for 5 seconds
front_end.run(5000)

# clean yp the machine for the next application.
front_end.stop()
```


Design Considerations I: Finite resources per core

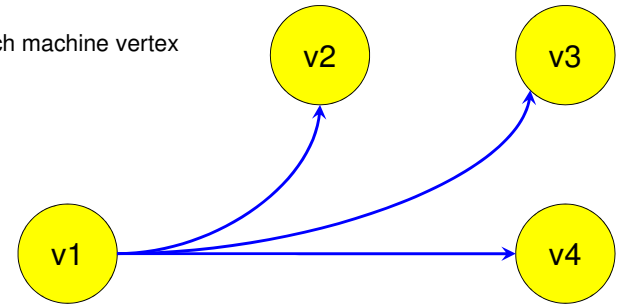
- The user's graph will be mapped to the cores of the SpiNNaker machine
- Each core has finite resource:
  - Compute power
  - Local memory
  - Share of SDRAM capacity & bandwidth
  - Communications bandwidth for packets
- Where each vertex represents many atoms we ***partition*** each one into smaller pieces, so that one piece fits on one core:
  - **Application graph** maps to **Machine Graph**
  - Edges also split to maintain correct connectivity
- Merging of vertices NOT currently supported!





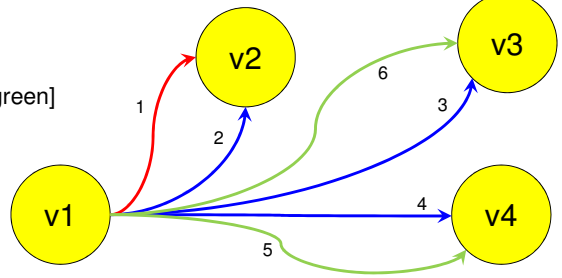
Design Considerations II: Dataflow between Vertices

- Consider the pattern of messages flowing from each vertex:
  - Case 1: Messages always go to the same set of targets
  - Case 2: Messages go to different targets at different times
- Case 1: Homogeneous data flow
  - e.g. spikes in neural simulation
- One **identifier** for each machine vertex

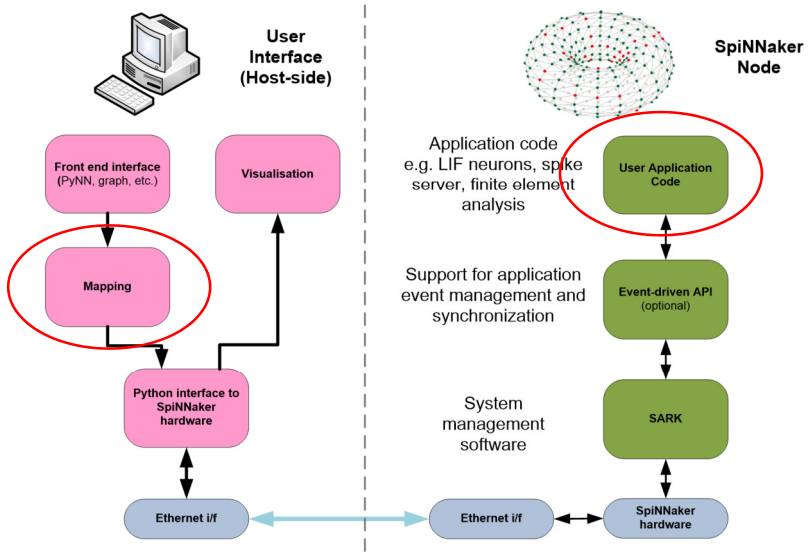


Design Considerations II: Dataflow between Vertices

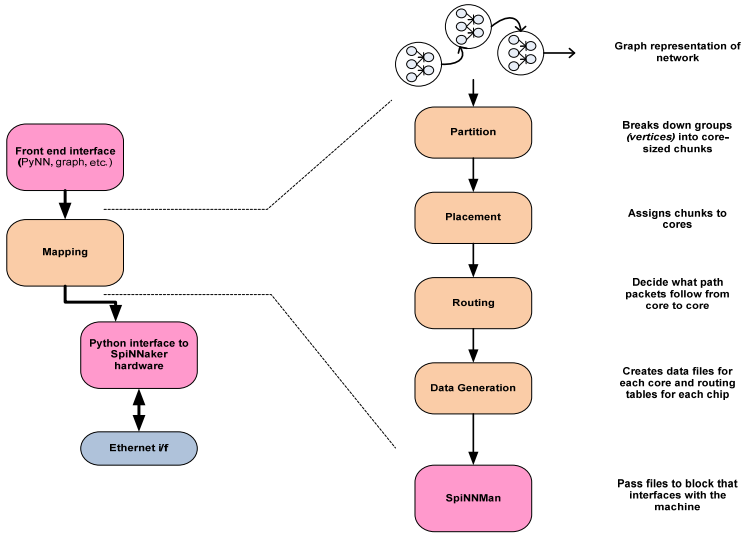
- Case 2: Data send to different targets at different times:
  - e.g. multi-layered perceptron, with forward and backward data flow
  - Useful when there are different modes of operation
- Group edges so that those in same mode are together
- A grouping is called a **partition**
- Assign a separate **identifier** for each pre-vertex/partition pair
- Six edges [1, 2, .., 6]
- Three partitions [red, blue, green]



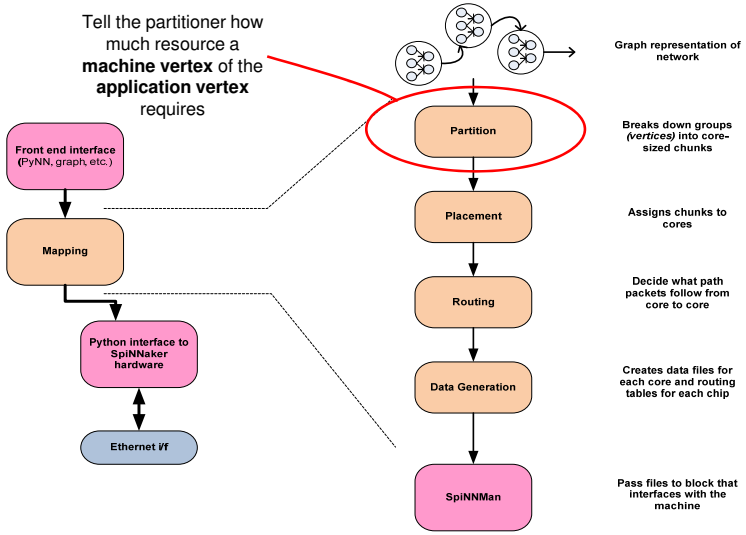
Software Stack



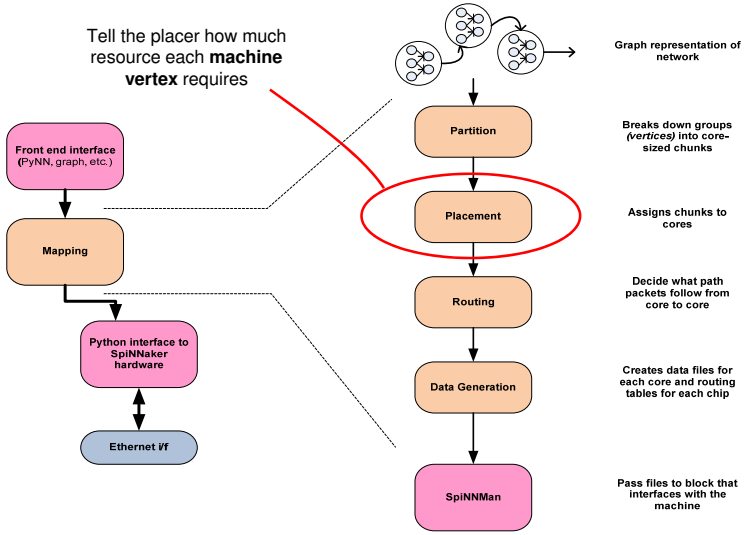
Where do you need to supply new information?



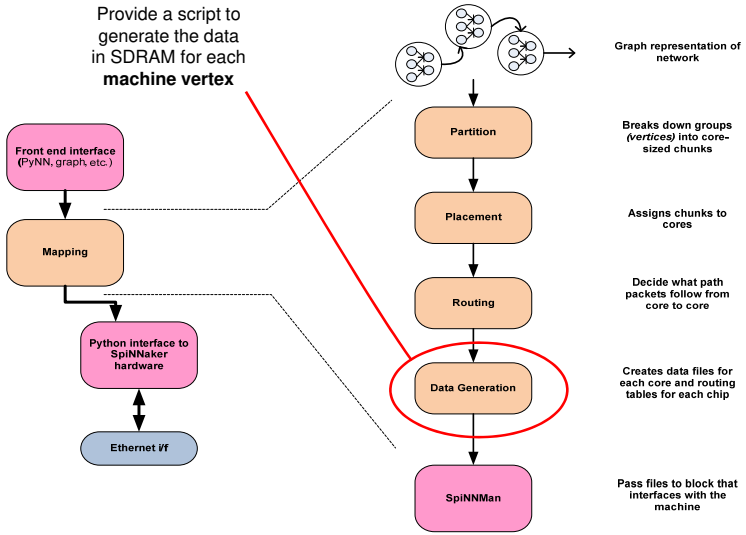
Where do you need to supply new information?



Where do you need to supply new information?



Where do you need to supply new information?



Data Spec. and Data Generation

- Each core running your application needs to generate its local data before it starts simulation
- We provide a simple ***virtual machine*** in which you can execute simple programs to generate this data
- This is the **Data Spec Executor (DSE)**
- The tools run code called the **Data Spec Generator (DSG)** that create a program (the specification or **spec**) for each core that is run by the DSG to generate its data

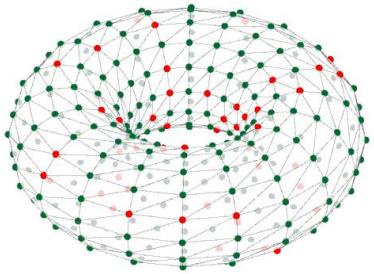
Summary

- It is useful to abstract any parallel application into the form of a graph with:
- Centres of computation (vertices)
- Connected by communication pathways (edges)

- **Application designer** must describe the computational elements and the communication types and plug those into our tools:
- Executables to run on SpiNNaker (typically written in C)
- Data specification, used to create each nodes data
- Describe resource requirements to allow tools to map networks to cores

- **User** can then specify application networks and run them using the Graph Front End.The tools handle :
- Mapping
- Routing table generation
- Data generation
- Loading
- Simulation
- Results gathering
- And other stuff ….

Introduction to the Graph Front End Functionality



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

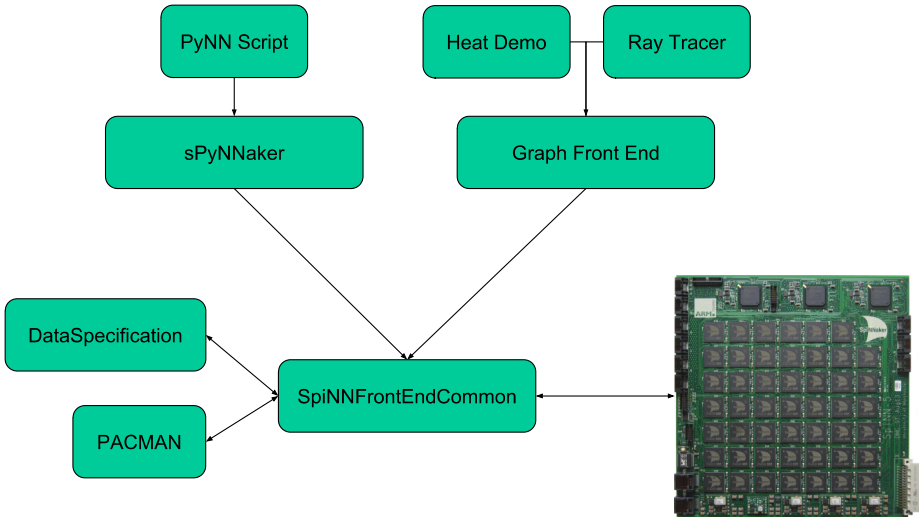
EPSRC

SpiNNaker

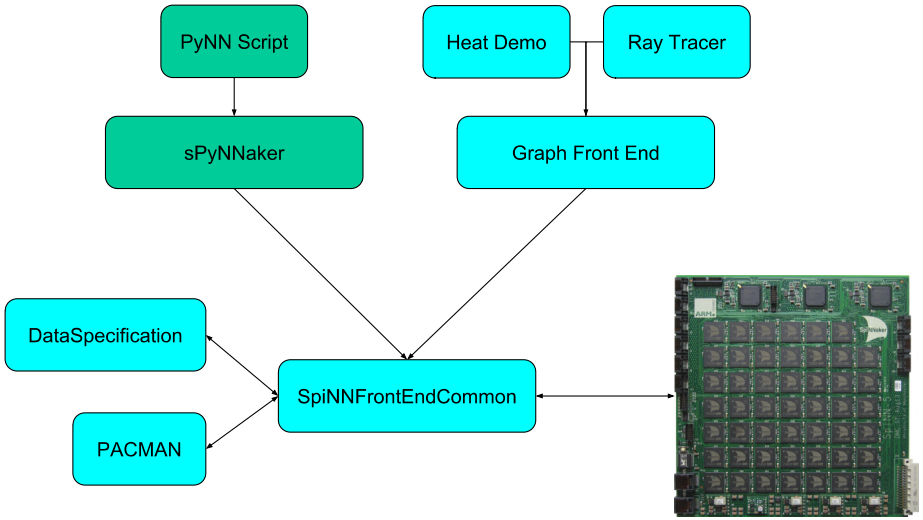
Contents

- The Graph Front End (GFE) interface
- Simple Usage of the GFE
- The Graph in the GFE
- Data Generation
- Binary Specification
- Writing and building simple C code

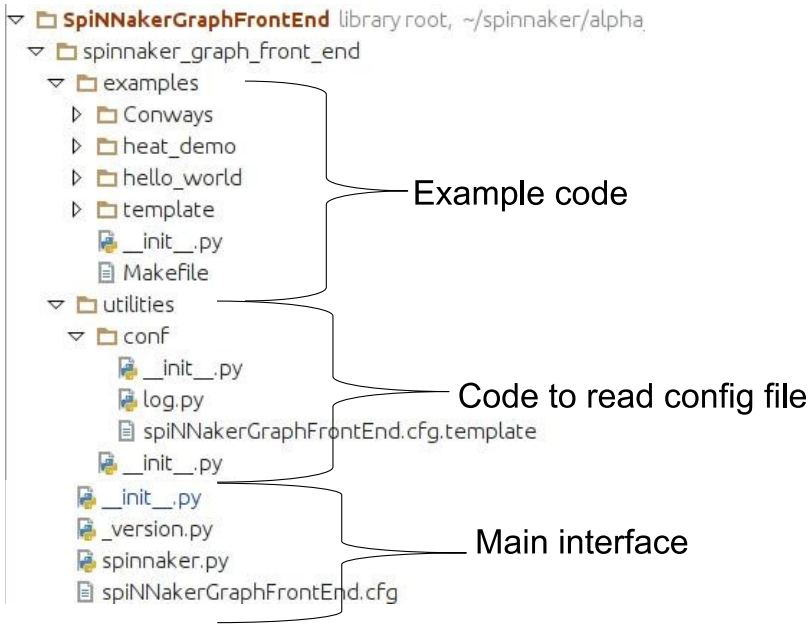
Software modules



Software modules covered here



GFE structure



Skeleton Functionality

```
import spinnaker_graph_front_end as front_end

# set up the front end
front_end.setup()

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```

ConfigurationException:
There needs to be a graph which contains at least one vertex for the tool chain to map anything.

Main interface functions

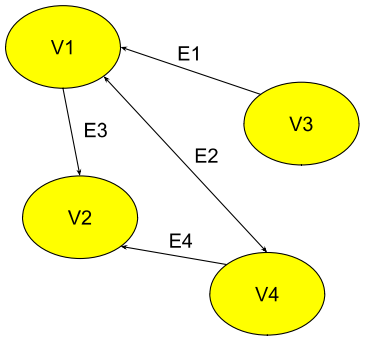
```
import spinnaker_graph_front_end as p
```

p.setup() Sets up the software stack so that it has read the configuration file and created whatever data objects are required.

p.run(duration) Runs the simulation for a given time period (microseconds).

p.stop() Closes down the application that is running on the SpiNNaker machine and does any housekeeping needed to allow the next application to run correctly.

PACMAN Graph



Has a 1:1 ratio between vertices and SpiNNaker core.

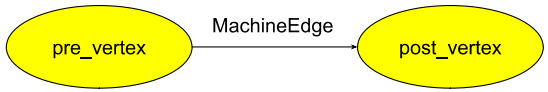
Basic script to add machine vertices into the graph

```
import spinnaker_graph_front_end as front_end

from spinnaker_graph_front_end.examples.Conways.conways_cell import \
  ConwayMachineCell

# set up the front end
front_end.setup()

for count in range(0, 800):
   front_end.add_machine_vertex_instance(
      ConwayMachineCell(label="cell{}".format(count)))

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```


Creating a new type of machine vertex

```
from pacman.model.graphs.machine.impl.machine_vertex import MachineVertex
from pacman.model.resources.resource_container import ResourceContainer

class ConwayMachineCell(MachineVertex):
  """ Cell which represents a cell within the 2D grid
  """

  def __init__(self, label):

    # construct the resources this cell uses and instantiate superclass
    resources = ResourceContainer()
    MachineVertex.__init__(self, resources, label)
```


Adding edges to the machine graph



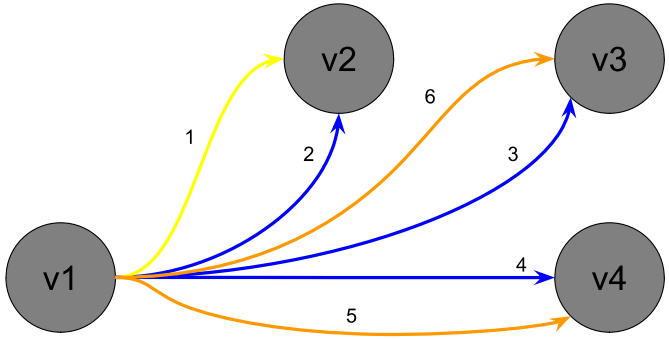
1. The main edge type available is MachineEdge.
2. This can be extended to add application specific data into.
3. Most important inputs are:
   i. **pre_vertex**: The source of the edge.
   ii. **post_vertex**: The destination of the edge.
4. Every edge in a graph is associated with a **partition_id**.

Basic Script adding edges

```
import spinnaker_graph_front_end as front_end
.......

# build and add vertices to the graph
vertices = list()
for count in range(0, 100):
  vertices.append(ConwayMachineCell("cell{}".format(count)))
  front_end.add_machine_vertex_instance(vertices[count])

# build an edge between two vertices
front_end.add_machine_edge_instance(
   MachineEdge(verts[0], verts[1]), "State")

front_end.run(5000)

front_end.stop()
```

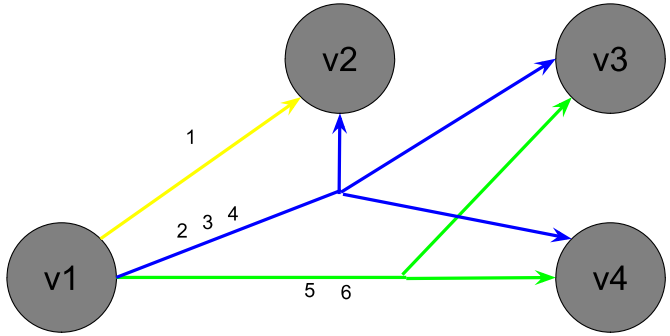
Partition id

Adding edges to the application graph: Partitions



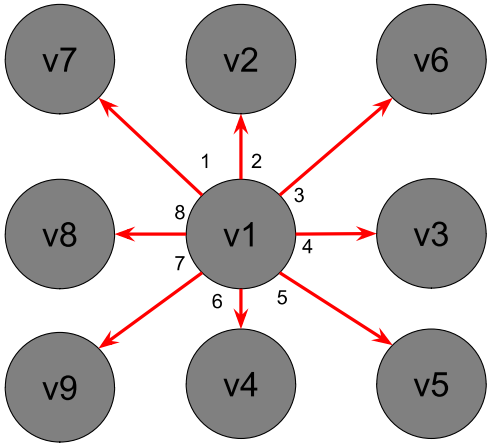
Edge 1 resides in partition A
Edges 2,3 and 4 reside in partition B
Edges 5 and 6 reside in partition C

Adding edges to the application graph: Partitions



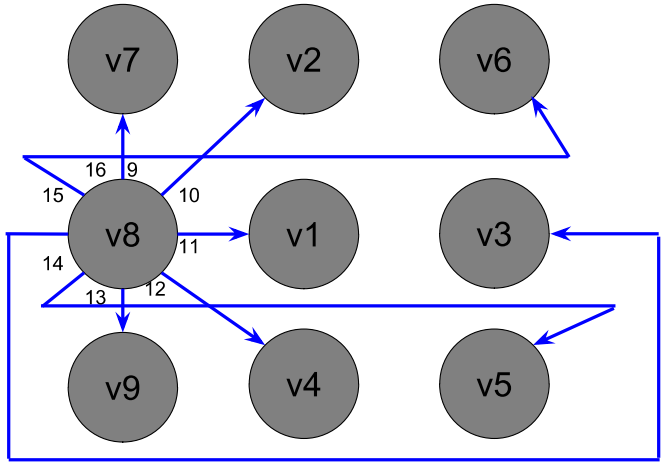
Edge 1 transmits information about **hotdogs**.
Edges 2,3 and 4 transmits information about **cats**.
Edges 5 and 6 transmits information about **bacon**.

Conways: partitions.



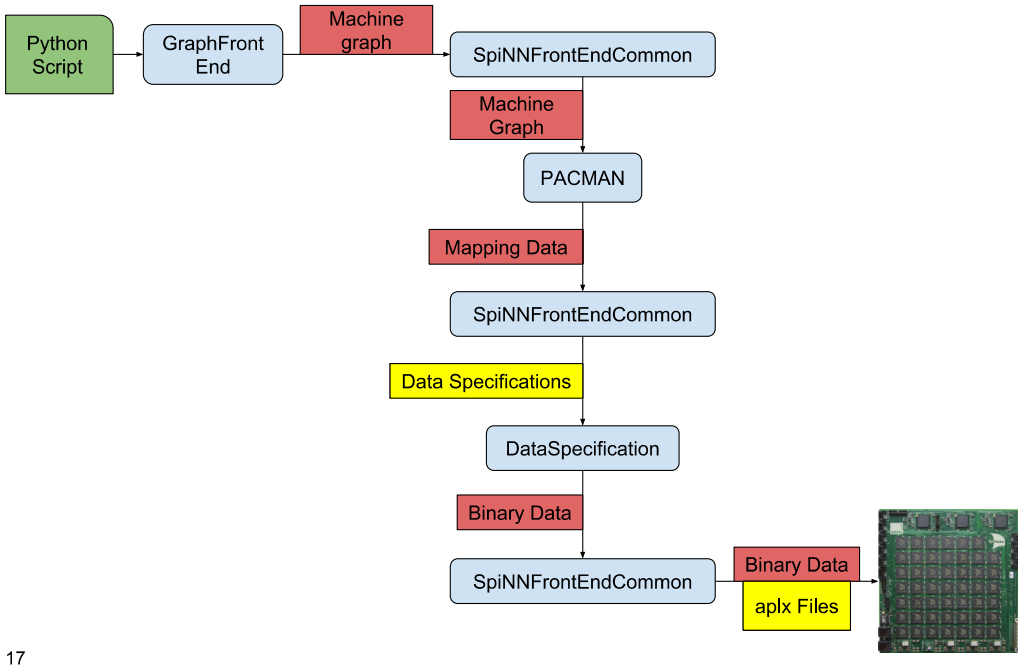
Edges 1,2,3,4,5,6,7,8 transmits v1's **state** data.

Conways: partitions.



Edges 9,10,11,12,13,14,15,16 transmits v8's **state** data.

Workflow of the GFE



Data generation

1. Converts application data within a vertex into data stored on the SpiNNaker machine via SDRAM.
2. Supports separating the SDRAM into data regions
3. Supports writing data as scalars, arrays etc.
4. Common commands:
   i. reserve_memory_region()
   ii. switch_write_focus()
   iii. write_value()
   iv. write_array()
   v. comment()
   vi. close_spec()

Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

    # Reserve SDRAM space for memory areas:
    spec.reserve_memory_region(
        region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
    spec.reserve_memory_region(
        region=1, size=8, label="inputs")
```


Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

    # Reserve SDRAM space for memory areas:
    spec.reserve_memory_region(
        region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
    spec.reserve_memory_region(
        region=1, size=8, label="inputs")

    # add simulation.c interface data
    spec.switch_write_focus(0)
    spec.write_array(simulation_utilities.get_simulation_header_array(
        self.get_binary_file_name(), machine_time_step, time_scale_factor))
```

Needed for simulation.c

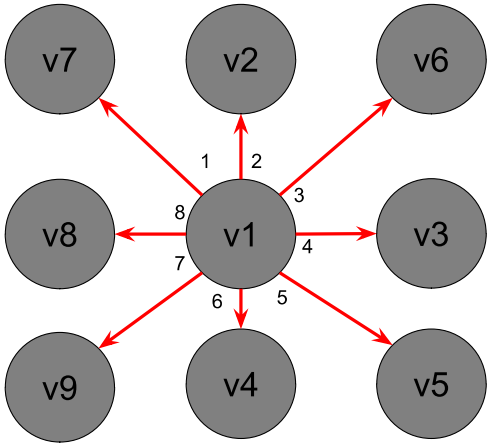
Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, Iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
    region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
    region=1, size=8, label="inputs")

# add simulation.c interface data
spec.switch_write_focus(0)
spec.write_array(simulation_utilities.get_simulation_header_array(
    self.get_binary_file_name(), machine_time_step, time_scale_factor))

# application specific data items
spec.switch_write_focus(region=1)
spec.comment("writing initial state for this conway element \n")
spec.write_value(data=self._state)
```


Data generation

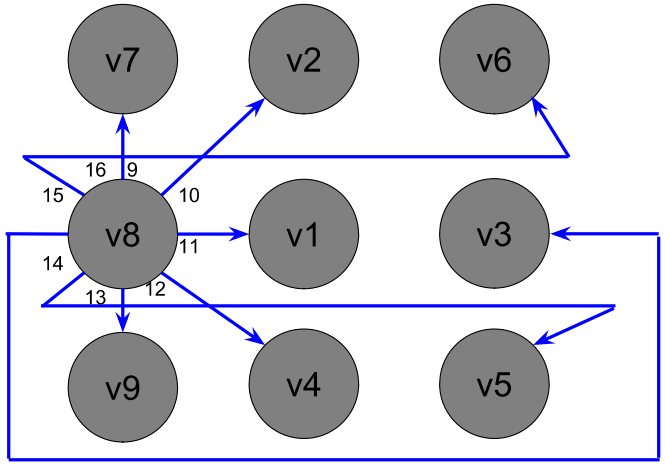
```
...
spec.comment("writing initial state for this conway element \n")
spec.write_value(data=self._state)

# write the routing key needed for my transmission
spec.comment("writing the routing key needed to transmit my state \n")
spec.write_value(data=routing_info.get_first_key_from_pre_vertex(self, "State"))

# close the spec
spec.comment("closing the spec \n")
spec.close_spec()
```


Conways: partitions.



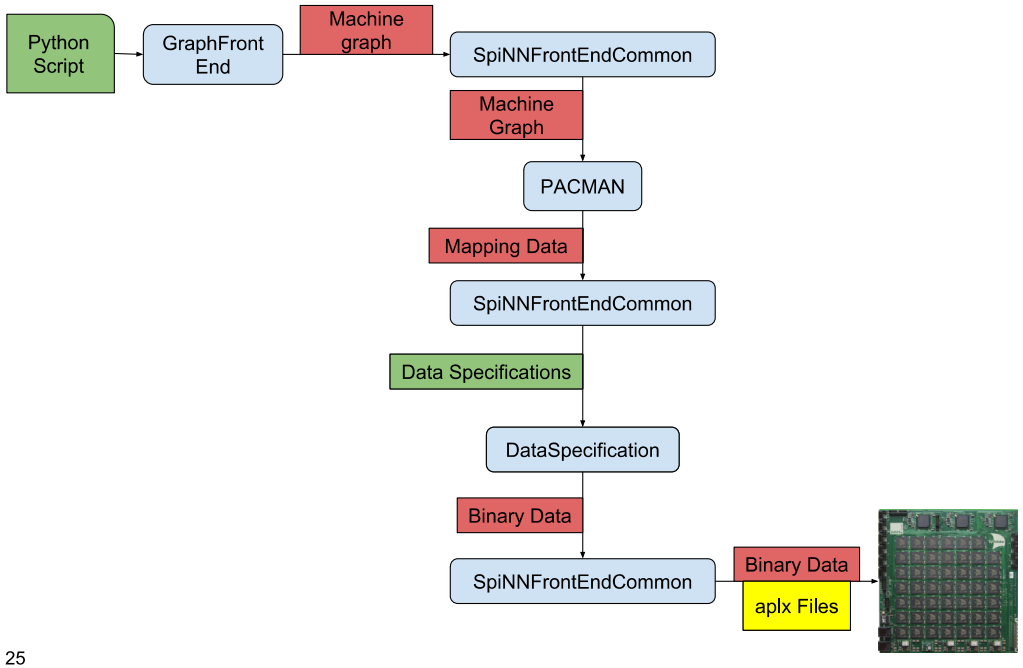
Edges 1,2,3,4,5,6,7,8 transmits with routing key 0.

Conways: partitions.



Edges 9,10,11,12,13,14,15,16 transmit with routing key 1.

Workflow of the GFE



Binary Executables

AbstractHasAssociatedBinary

```
def get_binary_file_name(self)
```

AbstractStartsSynchronized

- After loading binary, CPU state will be in SYNC0

AbstractBinaryUsesSimulationRun(AbstractStartsSynchronized)

- The binary uses the simulation environment provided by the tools

Binary Executables

```
from pacman.model.graphs.machine.impl.machine_vertex import MachineVertex
from pacman.model.resources.resource_container import ResourceContainer
from spinn_front_end_common.abstract_models.abstract_has_associated_binary\
    import AbstractHasAssociatedBinary
from spinn_front_end_common.abstract_models.abstract_binary_uses_simulation_run\
    import AbstractBinaryUsesSimulationRun

class ConwayMachineCell(
        MachineVertex, AbstractHasAssociatedBinary,
        AbstractBinaryUsesSimulationRun):
    """ Cell which represents a cell within the 2D grid
    """

    def __init__(self, label):

        # construct the resources this cell uses and instantiate superclass
        resources = ResourceContainer()
        MachineVertex.__init__(self, resources, label)

    def get_binary_file_name(self):

        # return the binary name of this vertex
        return "conways.aplx"
```


Linking Aplx files to Python

1. Compiled version of c code.
2. This code runs on SpiNNaker.
3. Is linked to your python classes through get_binary_file_name(self) of the vertex
4. How to write event driven C code for SpiNNaker is discussed in **Event Driven Simulations**.
5. We will cover the interfaces provided by the SpiNNFrontEndCommon (SFEC) module for the c code.

Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {

   // get address of simulation data
   address_t address = data_specification_get_data_address();
```

29

Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {

   // get address of simulation data
   address_t address = data_specification_get_data_address();

   // get the address of the system region
   address_t system_region = data_specification_get_region(0, address);
```

30

Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {
   ...
   if (!simulation_initialise(
         system_region, APPLICATION_NAME_HASH,
         &timer_period, &simulation_ticks,
         &infinite_run, SDP_PRIORITY,
         NULL, NULL)) {
      log_error("Error in initialisation - exiting!");
      rt_error(RTE_SWERR);
   }
```

31

Example c code

```
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {
   ...

   // Set timer_callback period
   spin1_set_timer_tick(timer_period);

   // Set timer_callback
   spin1_callback_on(TIMER_TICK, timer_callback, TIMER_PRIORITY);

   // Set time to UINT32 MAX to wrap around to 0 on the first timestep
   time = UINT32_MAX;

   simulation_run();
}
```

32

```
# gets the address where all the data for this core is stored.
address_t data_specification_get_data_address();
```

```
# gets the address of the start of a given data region
address_t data_specification_get_region(uint32_t region, address_t data_address)
```

```
# sets up the system to interact with the SFEC simulation control functionality.
bool simulation_initialise(
        address_t address, uint32_t expected_application_hash,
        uint32_t* timer_period, uint32_t *simulation_ticks_pointer,
        uint32_t *infinite_run_pointer, int sdp_packet_callback_priority,
        prov_callback_t provenance_function, address_t provenance_data_address)
```

```
# main entrance for the event driven nature of the SpiNNaker machine
void simulation_run()
```

Example c code

```
// Callbacks
void timer_callback(uint unused0, uint unused1) {

   // check if the simulation has run to completion
   if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
      simulation_exit();
   }
   time++;
}
```

```
# Used once you have finished your simulation
void simulation_exit()
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/
```

Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/

APP = conways

SOURCES = conways.c
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/

APP = conways

SOURCES = conways.c

include $(SPINN_DIRS)/make/Makefile.SpiNNFrontEndCommon
```


Summary

1. How to use the GFE interface.
2. The machine graph supported by the GFE.
3. Adding vertices, edges and partitions to the machine graph.
4. Data Specification for the graph.
5. Binary Specification.
6. Building and making basic C code.

6th SpiNNaker Workshop

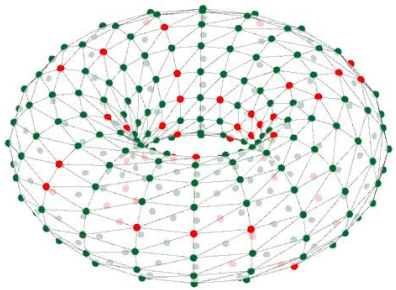
Day 3

September 7th 2016

Time	Session	Presenter
09:00	Simple data I/O and visualisation	ABS (SD)
10:00	Lab time (coffee at 10:30)	
11:00	Maths & fixed point libraries	MH
12:00	Lunch	
13:00	Adding new neuron models	AGR/MH
14:00	Connecting SpiNNaker to external devices	ABS (DRL)
14:30	Lab time (coffee at 15:00)	
16:30	Close	

Manchester, UK

Simple Data I/O and visualisation



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

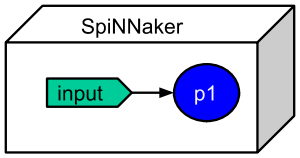
SpiNNaker

Contents

Standard PyNN support (Summary)

- Supports post execution gathering of certain attributes:
  - aka transmitted spikes, voltages etc.

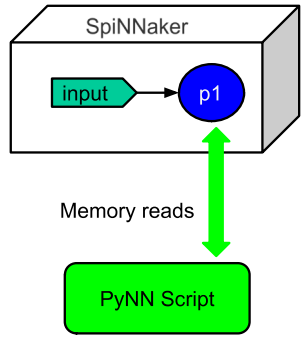
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
p1.record()
p1.record_v()
```



Standard PyNN support (Summary)

- Supports post execution gathering of certain attributes:
  - aka transmitted spikes, voltages etc.

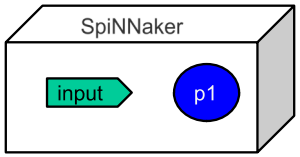
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
p1.record()
p1.record_v()
p.run(5000)
spikes = p1.getSpikes()
v = p1.get_v()
```



Standard PyNN support (Summary)

- Supports spike sources of:
  - Spike Source Array, Spike source poisson.

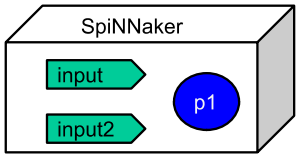
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
```



Standard PyNN support (Summary)

- Supports spike sources of:
  - Spike Source Array, Spike source poisson.

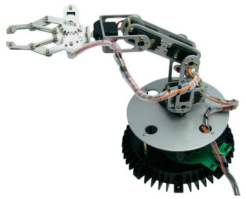
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
            {'spike_times': [0]}, label="input")
input2 = p.Population(1, p.SpikeSourcePoisson,
            {'rate':100, 'duration':50}, label='input2')
```



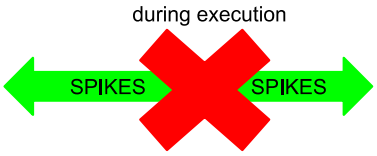
Standard PyNN support (Summary)

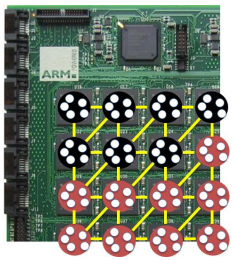
Restrictions

1. Recorded data is stored on SDRAM on each chip.
2. Data to be injected has to be known up-front, or rate based.
3. No support for closed loop execution with external devices.



http://www.conrad.com/ce/en/product/191516/Arexx-RA1-PRO-Metallic-Robot-Arm



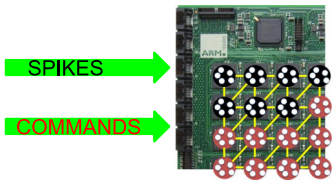


External Device Plugin

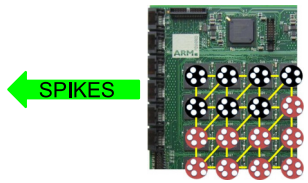
Why? what?

1. Contains functionality for PyNN scripts.
2. Not official PyNN!!!

What does it Includes?

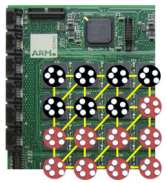


1. Live injection of events and commands into SpiNNaker



2. Live streaming of events from populations.

SPIKES via spinnLink interface



3. External devices support: Covered at 14:00

External Device Plugin

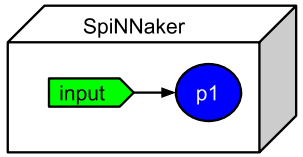
Caveats:

- Injection and live output currently only usable only with the ethernet connection,
- Limited bandwidth of:
  - A small number of spikes per millisecond time step, per ethernet,
  - Shared with both injection and live output,
- Best effort communication,
- Has a built in latency,
- Spinnaker commands not supported by other simulators,
- Loss of cores for injection and live output support,
- You can only feed a live population to one place.

Injecting spikes into PyNN scripts

PyNN script changes

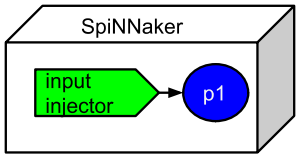
```
import pyNN.spiNNaker as p

p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                     {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
# loop(synfire connection)
loop_forward = list()
for i in range(0, n_neurons - 1):
        loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward, Frontend.FromListConnector(loop_forward))
```



Injecting spikes into PyNN scripts

PyNN script changes: Declaring an injector population

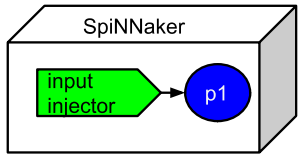
```
import pyNN.spiNNaker as p
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input_injector = p.Population(1, ExternalDevices.SpikeInjector,
                     {'port':95768}, label="injector)
input_proj = p.Projection(input_injector, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
# loop(synfire connection)
loop_forward = list()
for i in range(0, n_neurons - 1):
        loop_forward.append((i, (i + 1) % n_neurons, weight_to_spike, 3))
Frontend.Projection(pop_forward, pop_forward, Frontend.FromListConnector(loop_forward))
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
```

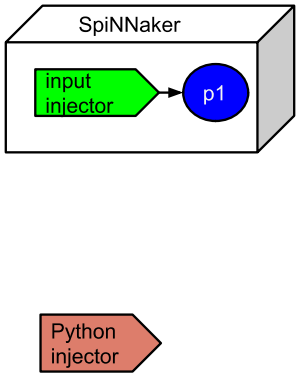


Python injector

Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

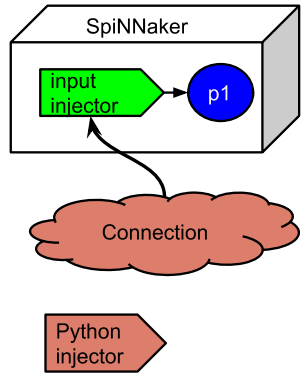
```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

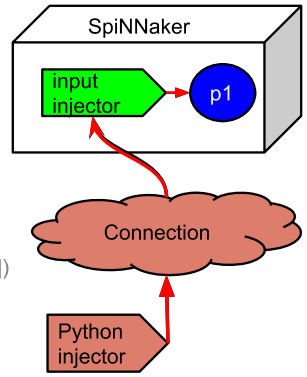
```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python injector connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
    receive_labels=None, local_port=19996, send_labels=["spike_sender"])
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up python injector

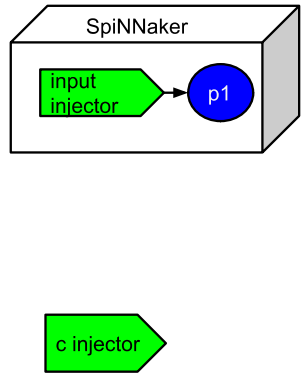
```
.....
# create python injector
def send_spike(label, sender):
        sender.send_spike(label, 0, send_full_keys=True)
# import python injector connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python injector connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
        receive_labels=None, local_port=19996, send_labels=["spike_sender"])
# register python injector with injector connection
live_spikes_connection.add_start_callback("spike_sender",  send_spike)
p.run(500)
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up c injector

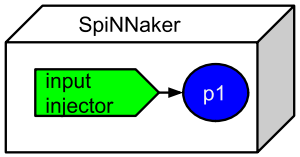
```
.....
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
```



Injecting spikes into PyNN scripts

PyNN script changes: Setting up c injector

```
.....
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
# import c injector connection
#include<SpynnakerLiveSpikeConnection.h>
```

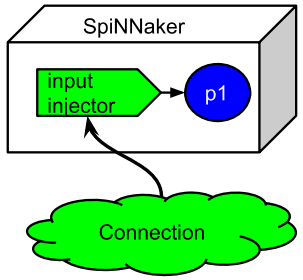


c injector

Injecting spikes into PyNN scripts

PyNN script changes: Setting up c injector

```
.....
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
# import c injector connection
#include<SpynnakerLiveSpikeConnection.h>
# set up c injector connection
SpynnakerLiveSpikesConnection live_spikes_connection =
SpynnakerLiveSpikesConnection(
        receive_labels=None, local_port=19996, send_labels=[“spike_sender”])
```

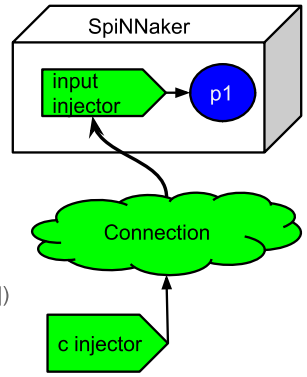


c injector

Injecting spikes into PyNN scripts

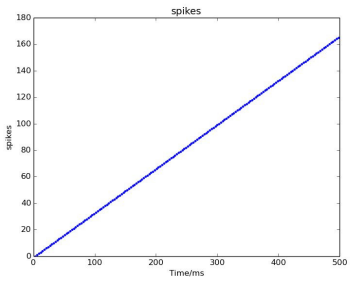
PyNN script changes: Setting up c injector

```
.....
# create c injector
void send_spike(str label, SpynnakerLiveSpikeConnection sender){
        sender.send_spike(label, 0, send_full_keys=True)  }
# import c injector connection
#include<SpynnakerLiveSpikeConnection.h>
# set up c injector connection
SpynnakerLiveSpikesConnection live_spikes_connection =
SpynnakerLiveSpikesConnection(
        receive_labels=None, local_port=19996, send_labels=[“spike_sender”])
# register c injector with injector connection
live_spikes_connection.add_start_callback(“spike_sender”,  send_spike)
```



Injecting spikes into PyNN scripts

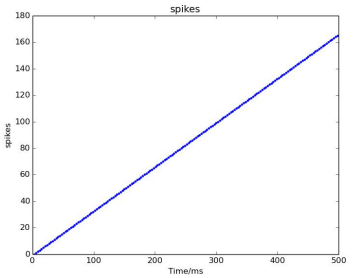
Behaviour with (SpikeSourceArray)



SAME!!!!!

BUT BORING!!!!

Behaviour with Live injection!



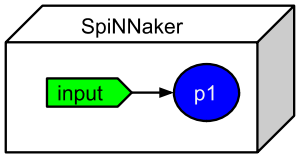
DEMO TIME!!! Injection

PYTHON DEMO!!!!

Live output from PyNN scripts

PyNN script changes: declaring live output population

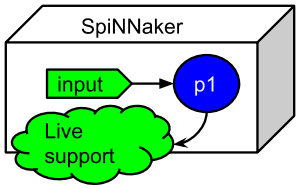
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
```



Live output from PyNN scripts

PyNN script changes: declaring live output population

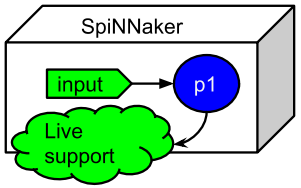
```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
p1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
                {'spike_times': [0]}, label="input")
input_proj = p.Projection(input, p1, p.OneToOneConnector(
  weights=5.0, delays=1))
# declare a live output for a given population.
import spynnaker_external_devices_plugin.pyNN as ExternalDevices
ExternalDevices.activate_live_output_for(p1)
```



Live output from PyNN scripts

PyNN script changes: python receiver

```
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
    for neuron_id in neuron_ids:
        print "Received spike at time {} from {}-{}"
            .format(time, label, neuron_id)
```

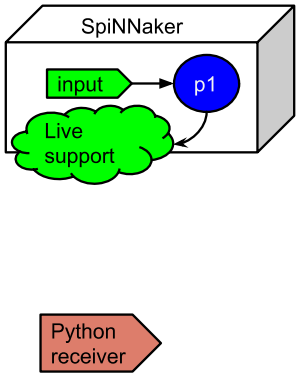


Python receiver

Live output from PyNN scripts

PyNN script changes: python receiver

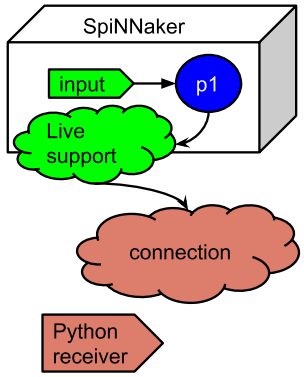
```
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print “Received spike at time {} from {}-{}”
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
```



Live output from PyNN scripts

PyNN script changes: python receiver

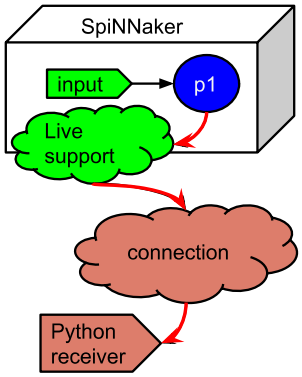
```
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print “Received spike at time {} from {}-{}”
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
    receive_labels=[“receiver”], local_port=19995, send_labels=None)
```



Live output from PyNN scripts

PyNN script changes: python receiver

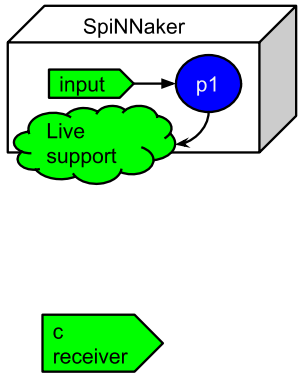
```
# declare python code when received spikes for a timer tick
def receive_spikes(label, time, neuron_ids):
        for neuron_id in neuron_ids:
                print “Received spike at time {} from {}-{}”
                        .format(time, label, neuron_id)
# import python live spike connection
from spynnaker_external_devices_plugin.pyNN.connections.\
spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
# set up python live spike connection
live_spikes_connection = SpynnakerLiveSpikesConnection(
        receive_labels=[“receiver”], local_port=19995, send_labels=None)
# register python receiver with live spike connection
live_spikes_connection.add_receive_callback(“receiver”, receive_spikes)
p.run(500)
```



Live output from PyNN scripts

PyNN script changes: c receiver

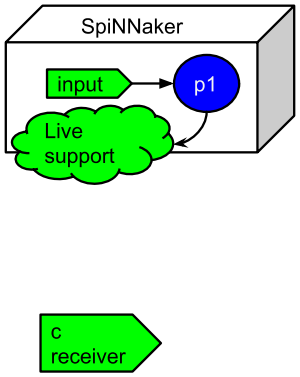
```
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf (“Received spike at time %d from %s-%d”,
                time, label, neuron_id.next());   } }
```



Live output from PyNN scripts

PyNN script changes: c receiver

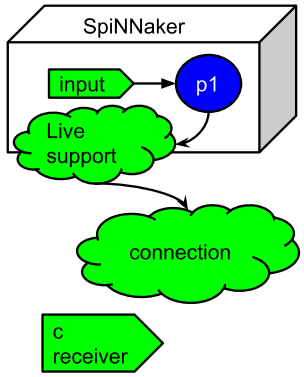
```
.....
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf (“Received spike at time %d from %s-%d”,
                time, label, neuron_id.next());   } }
# import c live spike connection
# include<SpynnakerLiveSpikesConnection.h>
```



Live output from PyNN scripts

PyNN script changes: c receiver

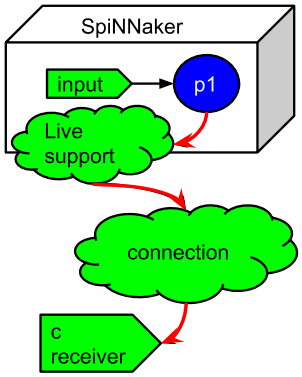
```
.....
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf (“Received spike at time %d from %s-%d”,
                time, label, neuron_id.next());   } }
# import c live spike connection
# include<SpynnakerLiveSpikesConnection.h>
# set up c live spike connection
SpynnakerLiveSpikesConnection live_spikes_connection =
 SpynnakerLiveSpikesConnection(
        receive_labels=[“receiver”], local_port=19995, send_labels=None);
```



Live output from PyNN scripts

PyNN script changes: c receiver

```
.....
# declare c code when received spikes for a timer tick
void receive_spikes(str label, int time, vector<int> neuron_ids){
        for (int index =0; index < neuron_ids.size(); index ++) {
                printf (“Received spike at time %d from %s-%d”,
                time, label, neuron_id.next());   } }
# import c live spike connection
# include<SpynnakerLiveSpikesConnection.h>
# set up c live spike connection
SpynnakerLiveSpikesConnection live_spikes_connection =
 SpynnakerLiveSpikesConnection(
        receive_labels=[“receiver”], local_port=19995, send_labels=None);
# register c receiver with live spike connection
live_spikes_connection.add_receive_callback(“receiver”, receive_spikes);
```



DEMO TIME!!! receive live spikes

PYTHON DEMO!!!!

Visualisation

How current supported visualisations work:

1. Uses the live output functionality as discussed previously.
2. Uses the c based receiver and is planned to be open source for users to augment with their own special visuals.
3. Currently contains raster plot support.

Visualisation

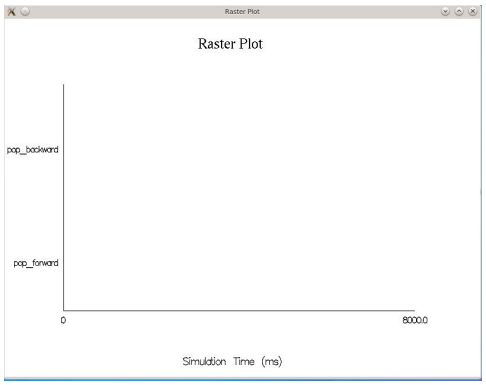
```
cspc277-visualiser-) make -f Makefile.linux
cspc277-visualiser-) ..............
cspc277-visualiser-) ./vis -colour_map test_data/spikeio_colours
cspc277-visualiser-)
awaiting tool chain hand shake to say database is ready
```

Input parameters:

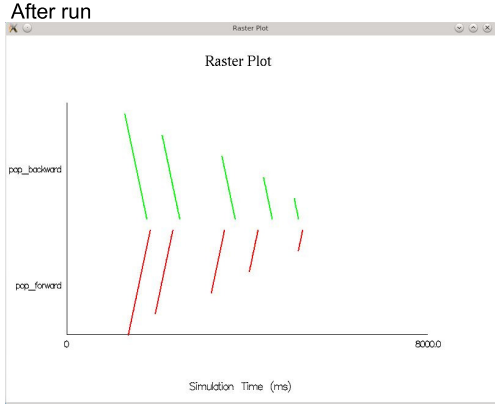
- -colour_map
  - Path to a file containing the population labels to receive, and their associated colours
- -hand_shake_port
  - optional port which the visualiser will listen to for database hand shaking
- -database
  - optional file path to where the database is located, if needed for manual configuration
- -remote_host
  - optional remote host, which will allow port triggering

Visualisation

Before run



After run



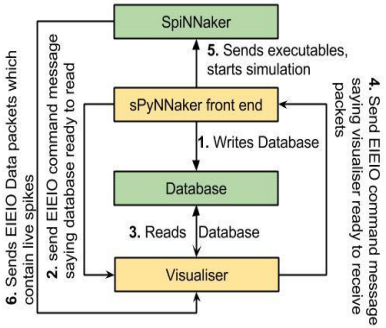
DEMO TIME!!! visualiser and injection of spikes

PYTHON DEMO!!!!

Technical Detail!!!

Notification protocol under the hood!

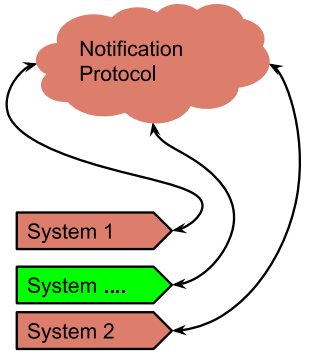
- Everything so far uses the notification protocol.
- It supplies data to translate spikes into population ids.
- If you have more than 1 system running to inject and/or receive, then you need to register this with the notification protocol.



Injecting spikes into PyNN scripts

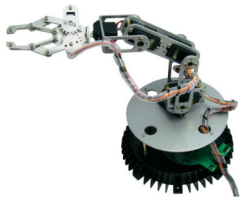
PyNN script changes:
registering a system to the notification protocol

```
.....
# register socket addresses for each system
p.register_database_notification_request(
    hostname=”local_host”
    notify_port=19990,
    ack_port=19992)
p.register_database_notification_request(
    hostname=”local_host”
    notify_port=19993,
    ack_port=19987)
p.register_database_notification_request(
    hostname=”local_host”
    notify_port=19760,
    ack_port=19232)
```

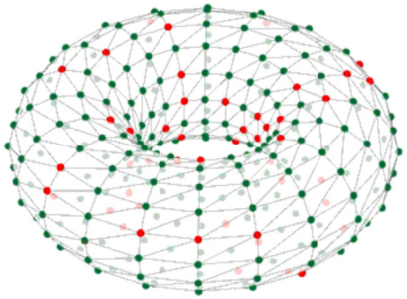


Thanks for listening

Any questions?!



Fixed-Point Maths and Libraries



Michael Hopkins

SpiNNaker Workshop, 7th September 2016

Overview

1. Numerical calculation on SpiNNaker
2. ISO/IEC 18037 types and operations
3. A simple example
4. Some practical considerations
5. Libraries currently available
6. An example using the libraries
7. Using fixed-point to solve ODEs
8. Future directions

Numerical calculation on SpiNNaker

- No floating point hardware on SpiNNaker
- Software floating point available but too slow for most use cases (and larger binaries)
- Until recently, has needed hand-coded fixed point types and manipulations
- This approach not transparent so can be prone to maintenance issues & mysterious bugs
- More difficult than necessary for developers to translate algorithms into source code
- ISO draft 18037 for fixed point types and operations seen as a good solution

ISO 18037 types and operations

- Draft standard for native fixed point types & operations used like integer or floating point
- Currently only available on GNU toolchain >= 4.7 and ARM target architecture
- 8-, 16-, 32 and 64-bit precisions all available in (un-)saturated and (un-)signed versions
- *accum* type is 32-bit 'general purpose real'; we support io_printf() with s16.15 & u16.16
- *fract* type is 16-bit in [0,1]; we support io_printf() with s0.15 & u0.16

Operations supported are:

- prefix and postfix increment and decrement operators (++, --)
- unary arithmetic operators (+, -, !)
- binary arithmetic operators (+, -, *, /)
- binary shift operators (<<, >>)
- relational operators (<, <=, >=, >)
- equality operators (==, !=)
- assignment operators (+=, -=, *=, /=, <<=, >>=)
- conversions to and from integer, floating-point, or fixed-point types

A simple example

```
#include <stdfix.h>

#define REAL accum
#define REAL_CONST( x ) x##k

REAL  a, b, c = REAL_CONST( 100.001 );
accum d = REAL_CONST( 85.08765 );

int c_main( void )
{
   for( unsigned int i = 0; i < 50; i++ ) {

      a = i * REAL_CONST( 5.7 );

      b = a - i;

      if( a > d ) c  = a + b;
      else        c -= b;

      io_printf( IO_STD,
                 "\n i %u  a = %9.3k  b = %9.3k  c = %9.3k", i, a, b, c );
      }

   return 0;
}
```

Some practical considerations

- Range & precision e.g. for *accum* (s16.15) must have 0.000031 <= | x | <= 65536
- Still need to avoid divides in loops as these are slow on ARM architecture
- *saturated* types safe from overflow but significantly slower
- Need to remember that numerical precision is absolute rather than relative
- Literal constants require type suffix – simplest way is via macro REAL_CONST()
- Don't forget to `#include <stdfix.h>`
- Disciplined use of REAL and REAL_CONST() macros can parameterise entire code base
- Be careful to use the correct type suffix otherwise floating-point will be assumed

Libraries currently available - 1

1) *random.h* – suite of pseudo random number generators by MWH

Provides three high quality uniform generators of *uint32_t* values; Marsaglia's KISS 32 and KISS 64 and L'Ecuyer's WELL1024a.

- All three 'pass' the very stringent DIEHARD, dieharder and TestU01 test suites
- Trade-offs between speed, cycle length and equi-distributional properties
- Available in both simple-to-use form and with full user control over seeds

Have used these Uniform PRNGs as the basis for a set of Non-Uniform PRNGs including currently the following distributions:

- Gaussian
- Poisson (optimised for small rates at the moment)
- Exponential

...with more on the way. Let us know your requirements and we will try to help.

Libraries currently available - 2

2) *stdfix-full-iso.h* & *stdfix-math.h* – ISO & transcendental functions by DRL

Fill in the gaps in the GCC implementation of the ISO draft fixed point maths standard and some extensions:

- Standardised type conversions between fixed point representations
- Utility functions for all types i.e. abs(x), min(x), max(x), round(x), countls(x)
- Mechanism for automatically inferring the right argument type (uses GNU extension)

Fixed point replacements for essential floating point *libm* functions i.e. expk(x), sqrtk(x), logk(x), sink(x), cosk(x) and others such as atank(x), powk(x,y), 1/x on the way

- Hand-optimised for speed and accuracy on ARM architecture
- 10-30x faster than *libm* calls, hence feasible for use inside loops if necessary

An example using the libraries

```
accum           a, b, c, d;
uint32_t        r1;
unsigned fract  uf1;

init_WELL1024a_simp();  // need to initialise WELL1024a RNG before use

for( unsigned int i = 0; i < 22; i++ ) {

    r1 = WELL1024a_simp();                    // draw from Uniform RNG

    uf1 = (unsigned fract) ulrbits( r1 );     // convert to unsigned fract

// draw from Std Gaussian distribution using MARS64
    a = gaussian_dist_variate( mars_kiss64_simp, NULL );

// do some calculations on a and then log()
    b = logk( absk( a * REAL_CONST( 100.0 ) ) );

// sqrt() of value drawn from Exponential distribution using WELL1024a
    c = sqrtk( exponential_dist_variate( WELL1024a_simp, NULL ) );

    d = expk( (accum) ( i - 10 ) );       // exp() from -10 to 11

    io_printf( IO_STD, "\n i %4u
      uf1=[Uniform{*}]= %8.6R  a=[Gauss{*}]= %7.3k b=[ln(abs(100 a))]= %7.3k
      c=[sqrt(Exponential{*})]= %7.3k  d=[exp(i-10)]= %10.3k ", i, uf1, a, b, c, d );

    }
```

Using fixed-point to solve ODEs - 1

- Simulating neuron models usually means solving Ordinary Differential Equations (ODEs)
- This ranges from very easy (current input LIF has simple closed-form) solution to very challenging i.e. Hodgkin-Huxley with 4 state variables, nonlinear and very 'stiff' ODE
- Numerical calculations are required with a balance between accuracy & efficiency
- With care and attention to detail, fixed-point can be used to get very close to floating-point results. However, models with more complex behaviour are a significant challenge
- A new approach called *Explicit Solver Reduction* (ESR) makes this easier in many cases and is described in: Hopkins & Furber (2015), “Accuracy and Efficiency in Fixed-Point Neural ODE Solvers”, *Neural Computation* **27**, 1–35
- Good results found for Izhikevich neuron at real-time simulation speed & 1 ms time step

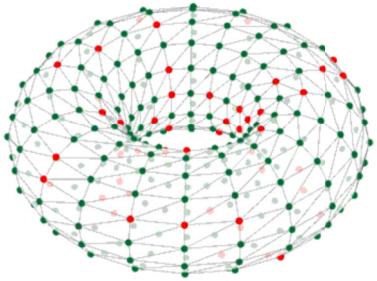
Using fixed-point to solve ODEs - 2

```
/*
   ESR algebraic reduction of the combination of Izhikevich neuron model and
   Runge-Kutta 2nd order midpoint method.  Hand-optimised interim variables and
   arithmetic ordering for balance between speed and accuracy.  See Neural Computation
   paper for more details.
*/
static inline void _rk2_kernel_midpoint( REAL h, neuron_pointer_t neuron,
                                         REAL input_this_timestep ) {
// to match Mathematica names
    REAL lastV1 = neuron->V;
    REAL lastU1 = neuron->U;
    REAL a = neuron->A;
    REAL b = neuron->B;

// generate common interim variables
    REAL pre_alph = REAL_CONST(140.0) + input_this_timestep - lastU1;
    REAL alpha = pre_alph
               + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * lastV1 ) * lastV1;
    REAL eta = lastV1 + REAL_HALF( h * alpha );

// could be represented as a long fract but need efficient mixed-arithmetic functions
    REAL beta = REAL_HALF( h * ( b * lastV1 - lastU1 ) * a );

// update neuron state
    neuron->V += h * ( pre_alph - beta
                     + ( REAL_CONST(5.0) + REAL_CONST(0.0400) * eta ) * eta );

    neuron->U += a * h * ( -lastU1 - beta + b * eta );
}
```

Future directions

- Optimise operations on differing fixed point types i.e. *accum* * *long fract*
- Add to *stdfix-math* (e.g. new argument types and special functions)
- Add to *random* (e.g. longer cycle uniform PRNG and more non-uniform distributions)
- New libraries such as probability distributions to allow Bayesian inference tools
- io_printf() to be extended to more types such as *long fract*, *unsigned long fract*
- Linear Algebra operations such as matrix multiply, SVD and other decompositions
- SpiNNaker architecture potentially good choice for massively parallel algorithms e.g. MCMC

Adding New Neuron Models



Andrew Rowley, Michael Hopkins

SpiNNaker Workshop, September 2016

European Research Council
Established by the European Commission

Human Brain Project

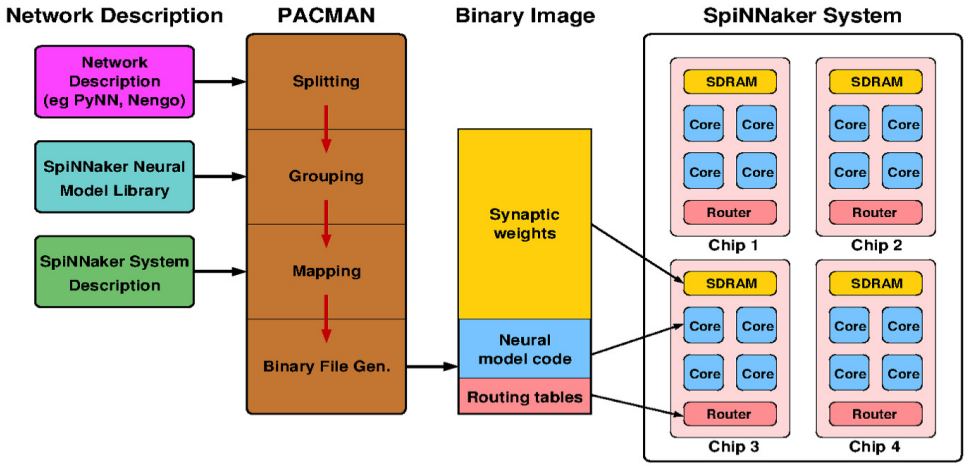
EPSRC

SpiNNaker

Required code separation

- Any new neuron model requires both C and Python code
- C code makes the actual executable (on SpiNNaker), Python code configures the setup and load phases (on the host)
- These are separate but must be perfectly coordinated
- In almost all cases, the C code will be solving an ODE which describes how the neuron state evolves over time and in response to input

Required code separation



We will first describe the C requirements...

C Data Structures and Parameters

- The parameters and state of a neuron at any point in time need to be stored in memory
- For each neuron, the C header defines the ordering and size of each stored value
- The C types can be standard integer and floating-point, or ISO draft standard fixed-point, as required (see later talk *Maths & fixed-point libraries*)
- There is also one global data structure which services all neurons on a core

So here is an example using the Izhikevich neuron...

Specific neuron model – data structure

```
#include "neuron-model.h"

//  Izhikevic neuron data structure defined in neuron_model_izh_curr_impl.h

typedef struct neuron_t {

// 'fixed' parameters - abstract units
   REAL   A;
   REAL   B;
   REAL   C;
   REAL   D;

// variable-state parameters
   REAL   V;          // nominally in [mV]
   REAL   U;

// offset current [nA]
   REAL   I_offset;

// private variable used internally in C code
   REAL   this_h;

} neuron_t;

...
```

Global data structure

```
...

/*
    Global data structure defined in neuron_model_izh_curr_impl.h
*/

typedef struct global_neuron_params_t {

// Machine time step in milliseconds
   REAL   machine_timestep_ms;

} global_neuron_params_t;
```

Implementing the state update

- Neuron models are typically described as systems of initial value ODEs
- At each time step, the internal state of each neuron needs to be updated in response to inherent dynamics and synaptic input
- There are many ways to achieve this; there will usually be a 'best approach' (in terms of balance between accuracy & efficiency) for each neuron model
- A recently published paper gives a lot more detail: Hopkins & Furber (2015), “Accuracy and Efficiency in Fixed-Point Neural ODE Solvers” , *Neural Computation* **27**, 1–35
- The key function will always be *neuron_model_state_update*(); the other functions are mainly to support this and allow debugging etc.

Continuing the example by describing the key interfaces...

Neuron model API

```
// pointer to a neuron data type - used in all access operations
typedef struct neuron_t* neuron_pointer_t;


// set the global neuron parameters
void neuron_model_set_global_neuron_params( global_neuron_params_pointer_t params );


// key function in timer loop that updates neuron state and returns membrane voltage
state_t neuron_model_state_update(
            input_t exc_input, input_t inh_input, input_t external_bias,
            neuron_pointer_t neuron );


// return membrane voltage (= first state variable) for a given neuron
state_t neuron_model_get_membrane_voltage( restrict neuron_pointer_t neuron );


// update the neuron structure to take account of a spike
void neuron_model_has_spiked( neuron_pointer_t neuron );


// print out neuron definition and/or state variables (for debug)
void neuron_model_print_parameters( restrict neuron_pointer_t neuron );
void neuron_model_print_state_variables( restrict neuron_pointer_t neuron );
```

Specific neuron model – key functions

```
/* simplified version of Izhikevic neuron code defined in neuron_model_izh_curr_impl.c  */


// key function in timer loop that updates neuron state and returns membrane voltage
state_t neuron_model_state_update(
             input_t exc_input, input_t inh_input, input_t external_bias,
             neuron_pointer_t neuron ) {

// collect inputs
    input_t input_this_timestep =
        exc_input - inh_input + external_bias + neuron->I_offset;

// most balanced ESR update found so far
    _rk2_kernel_midpoint( neuron->this_h, neuron, input_this_timestep );
    neuron->this_h = global_params->machine_timestep_ms;

// return the value of the membrane voltage
    return neuron->V;
}


// make the discrete changes to state after a spike has occurred
void neuron_model_has_spiked( neuron_pointer_t neuron ) {
    neuron->V  = neuron->C;       // reset membrane voltage
    neuron->U += neuron->D;       // offset 2nd state variable
}
```

Threshold Models – Interface and Implementation

Interface

```
// Pointer to threshold data type - used to access all operations
typedef struct threshold_type_t;


// Main interface function - determine if the value is above the threshold
static inline bool threshold_type_is_above_threshold(
        state_t value, threshold_type_pointer_t threshold_type );
```

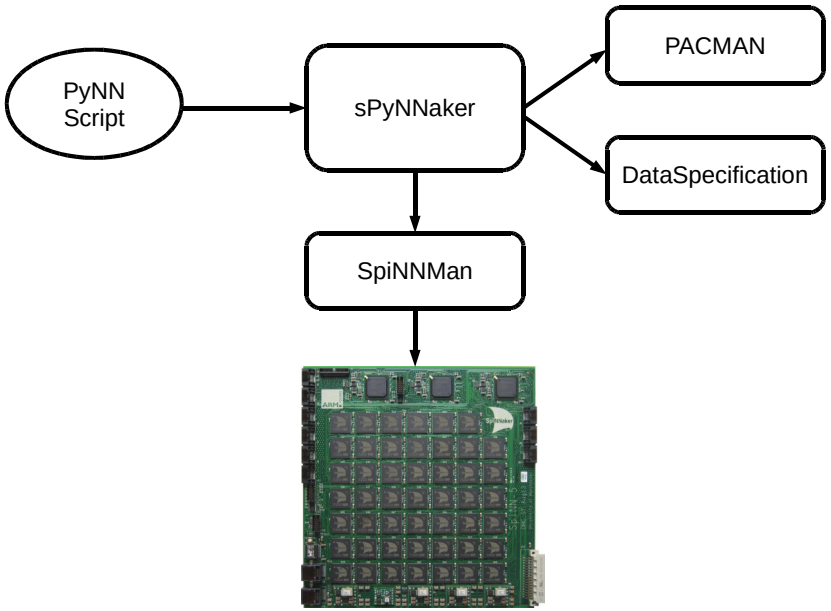
Static Threshold Implementation

```
typedef struct threshold_type_t {

    // The value of the static threshold
    REAL threshold_value;

} threshold_type_t;


static inline bool threshold_type_is_above_threshold(
        state_t value, threshold_type_pointer_t threshold_type ) {

    return REAL_COMPARE( value, >=, threshold_type->threshold_value );
}
```

Makefile

```
APP = my_model_curr_exp

# This is the folder where things will be built (this will be created)
BUILD_DIR = build/

# This is the neuron model implementation
NEURON_MODEL = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.c

# This is the header of the neuron model, containing the definition of neuron_t
NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/neuron_model_my_model_curr_exp.h

# This is the header containing the input type (current in this case)
INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h

# This is the header containing the threshold type (static in this case)
THRESHOLD_TYPE_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h

# This is the header containing the synapse shaping type (exponential in this case)
SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h

# This is the synapse dynamics type (in this case static i.e. no synapse dynamics)
SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c

# This includes the common Makefile that hides away the details of the build
include ../Makefile.common
```

Python Interface – Why?



Python Interface

```
from spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model \
    import AbstractNeuronModel

class NeuronModelIzh(AbstractNeuronModel):
    def __init__(self, n_neurons, a, b, c, d, v_init, u_init, i_offset):
        AbstractNeuronModel.__init__(self)
        self._n_neurons = n_neurons
```


Python Interface - Parameters

- Parameters can be:
  - Individual values
  - Array of values (one per neuron)
  - RandomDistribution
- Normalise Parameters
  - utility_calls.convert_param_to_numpy(param, n_neurons)

Python Interface – initializer

```
from spynnaker.pyNN.models.neuron.neuron_models.abstract_neuron_model \
    import AbstractNeuronModel

class NeuronModelIzh(AbstractNeuronModel):
    def __init__(self, n_neurons, a, b, c, d, v_init, u_init, i_offset):
        AbstractNeuronModel.__init__(self)
        self._n_neurons = n_neurons

        self._a = utility_calls.convert_param_to_numpy(a, n_neurons)
        self._b = utility_calls.convert_param_to_numpy(b, n_neurons)
        self._c = utility_calls.convert_param_to_numpy(c, n_neurons)
        self._d = utility_calls.convert_param_to_numpy(d, n_neurons)
        self._v_init = utility_calls.convert_param_to_numpy(v_init, n_neurons)
        self._u_init = utility_calls.convert_param_to_numpy(u_init, n_neurons)
        self._i_offset = utility_calls.convert_param_to_numpy(
            i_offset, n_neurons)
```


Python Interface – properties

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    @property
    def a(self):
        return self._a

    @a.setter
    def a(self, a):
        self._a = utility_calls.convert_param_to_numpy(a, self.n_atoms)

    @property
    def b(self):
        return self._b

    @b.setter
    def b(self, b):
        self._b = utility_calls.convert_param_to_numpy(b, self.n_atoms)

    ...
```

Python Interface – state initializers

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def initialize_v(self, v_init):
        self._v_init = utility_calls.convert_param_to_numpy(v_init, self.n_atoms)

    def initialize_u(self, u_init):
        self._u_init = utility_calls.convert_param_to_numpy(u_init, self.n_atoms)
```


Python Interface – parameters

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_neural_parameters(self):
        Return 8

    def get_parameters(self):
        return [
            # REAL a
            NeuronParameter(self._a, DataType.S1615),
            # REAL b
            NeuronParameter(self._b, DataType.S1615),
            # REAL c
            NeuronParameter(self._c, DataType.S1615),
            # REAL d
            NeuronParameter(self._d, DataType.S1615),
            # REAL v
            NeuronParameter(self._v_init, DataType.S1615),
            # REAL u
            NeuronParameter(self._u_init, DataType.S1615),
            # REAL I_offset
            NeuronParameter(self._i_offset, DataType.S1615),
            # REAL this_h
            NeuronParameter(self._machine_time_step / 1000.0, DataType.S1615)
        ]
```


Python Interface – global params

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_global_parameters(self):
        return 1

    @inject_items({"machine_time_step": "MachineTimeStep"})
    def get_global_parameters(self, machine_time_step):
        return [
            NeuronParameter(machine_time_step / 1000.0, DataType.S1615)
        ]
```


Python Interface - Injection

```
@inject_items({"machine_time_step": "MachineTimeStep"})
def get_global_parameters(self, machine_time_step):
```

- Some items can be “injected” from the interface
  - Specify a dictionary of parameter name to “type” to inject
  - Parameter is in addition to the interface
- Common types include:
  - MachineTimeStep
  - TimeScaleFactor
  - TotalRunTime

Python Interface – CPU usage

```
class NeuronModelIzh(AbstractNeuronModel):
    ...

    def get_n_cpu_cycles_per_neuron(self):

        # A bit of a guess
        return 150
```


Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """

    def __init__(self, n_neurons, v_thresh):
        AbstractThresholdType.__init__(self)
        self._n_neurons = n_neurons
        self._v_thresh = utility_calls.convert_param_to_numpy(
            v_thresh, n_neurons)
```

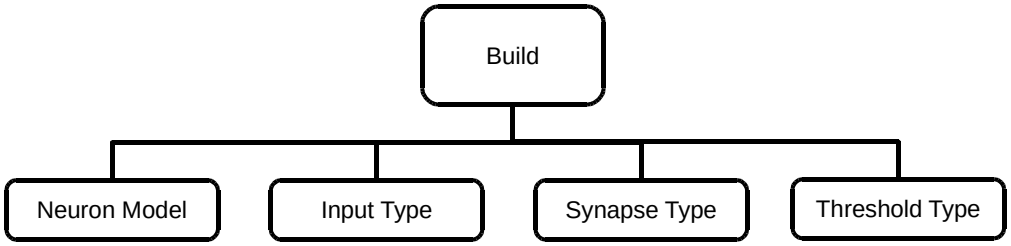

Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """
    ...

    @property
    def v_thresh(self):
        return self._v_thresh

    @v_thresh.setter
    def v_thresh(self, v_thresh):
        self._v_thresh = utility_calls.convert_param_to_numpy(
            v_thresh, self._n_neurons)
```


Python Interface – Threshold type

```
class ThresholdTypeStatic(AbstractThresholdType):
    """ A threshold that is a static value
    """
    ...

    def get_n_threshold_parameters(self):
        return 1

    def get_threshold_parameters(self):
        return [
            NeuronParameter(self._v_thresh, DataType.S1615)
        ]

    def get_n_cpu_cycles_per_neuron(self):

        # Just a comparison, but 2 just in case!
        return 2
```

Python Build



Python Build – Class Definition

```
from spynnaker.pyNN.models.abstract_models.abstract_population_vertex import \
    AbstractPopulationVertex

class IzkCurrExp(AbstractPopulationVertex):
```


Python Build

```
class IzkCurrExp(AbstractPopulationVertex):
    _model_based_max_atoms_per_core = 255
    default_parameters = {
        'a': 0.02, 'c': -65.0, 'b': 0.2, 'd': 2.0, 'i_offset': 0,
        'u_init': -14.0, 'v_init': -70.0, 'tau_syn_E': 5.0, 'tau_syn_I': 5.0}
```

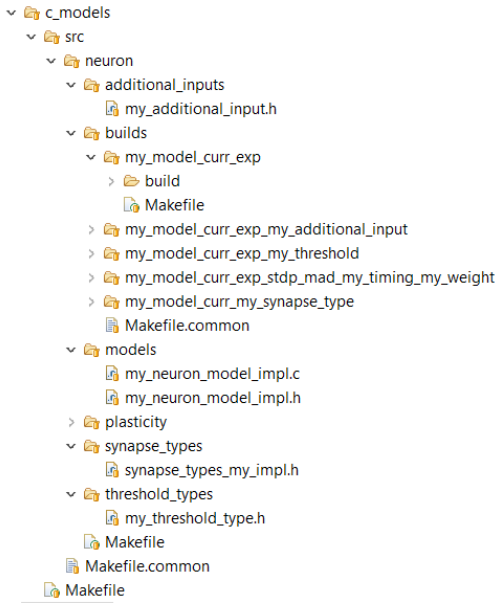

Python Build – initializer

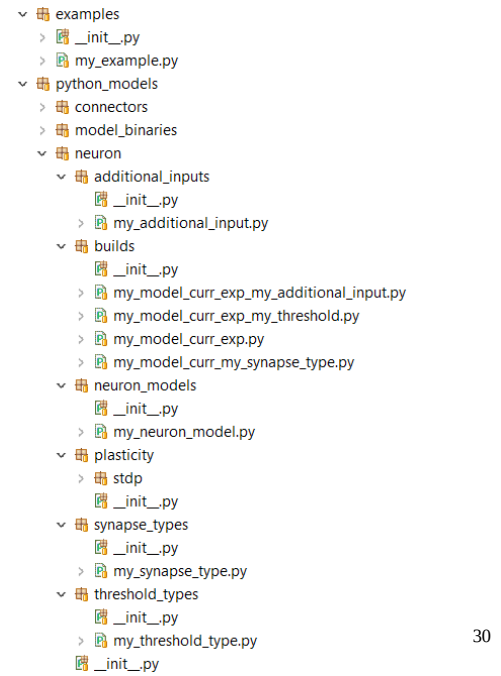
```
class IzkCurrExp(AbstractPopulationVertex):
    def __init__(
            self, n_neurons, spikes_per_second=None, ring_buffer_sigma=None,
            incoming_spike_buffer_size=None, constraints=None, label=None,
            a=default_parameters['a'], b=default_parameters['b'],
            c=default_parameters['c'], d=default_parameters['d'],
            i_offset=default_parameters['i_offset'],
            u_init=default_parameters['u_init'],
            v_init=default_parameters['v_init'],
            tau_syn_E=default_parameters['tau_syn_E'],
            tau_syn_I=default_parameters['tau_syn_I']):

        neuron_model = NeuronModelIzh(
            n_neurons, a, b, c, d, v_init, u_init, i_offset)
        synapse_type = SynapseTypeExponential(
            n_neurons, tau_syn_E, tau_syn_I)
        input_type = InputTypeCurrent()
        threshold_type = ThresholdTypeStatic(n_neurons, _IZK_THRESHOLD)

        AbstractPopulationVertex.__init__(
            self, n_neurons=n_neurons, binary="IZK_curr_exp.aplx", label=label,
            max_atoms_per_core=IzkCurrExp._model_based_max_atoms_per_core,
            spikes_per_second=spikes_per_second,
            ring_buffer_sigma=ring_buffer_sigma,
            incoming_spike_buffer_size=incoming_spike_buffer_size,
            model_name="IZK_curr_exp", neuron_model=neuron_model,
            input_type=input_type, synapse_type=synapse_type,
            threshold_type=threshold_type, constraints=constraints)
```

Python Build – max atoms

```
class IzkCurrExp(AbstractPopulationVertex):

    ...

    @staticmethod
    def set_model_max_atoms_per_core(new_value):
        IzhikevichCurrentExponentialPopulation.\
            _model_based_max_atoms_per_core = new_value

    @staticmethod
    def get_max_atoms_per_core():
        return IzkCurrExp._model_based_max_atoms_per_core
```


New Model Template

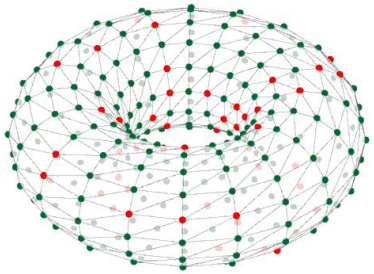




Using Your Model

```
import pyNN.spiNNaker as p
import python_models as new_models

my_model_pop = p.Population(
    1, new_models.MyModelCurrExp,
    {"my_parameter": 2.0,
     "i_offset": i_offset},
    label="my_model_pop")
```

External Devices



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

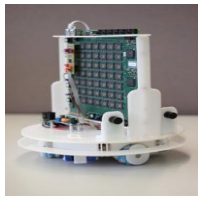
EPSRC

SpiNNaker

Contents

1. How to add external devices that communicate through the SpiNNaker Link into your PyNN scripts.
2. How to add external devices that communicate through the FPGA/SATA connector into your PyNN scripts.
3. How FPGA's are used within multi board systems.

Real time systems?



SpOmnibot
(Retinas & Motors)

FPGA connector

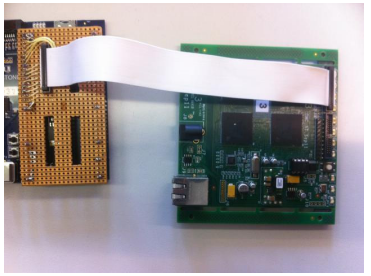
A Retina

A Osaka retina

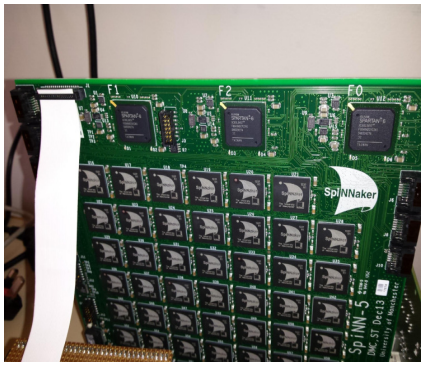
A Cochlea

How to connect devices to a spiNNaker board

Connect the device to the SpiNNaker link connector



connecting to a
spinn-3 Board

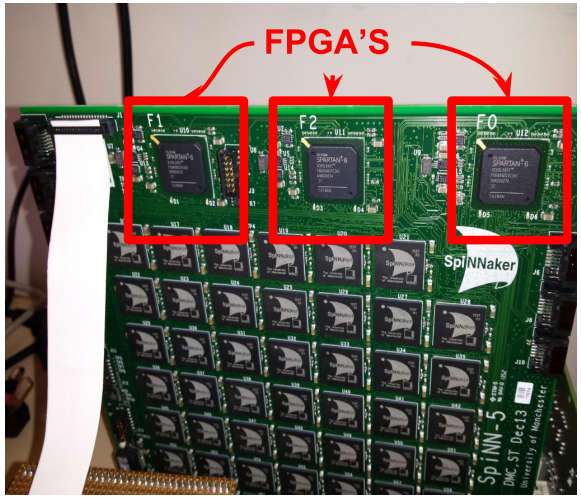


connecting to a
spinn-5 Board

FPGA Programming

1. The FPGA's need repogramming to support external device plugin.

2. This reprogramming is not done by the tools to date.



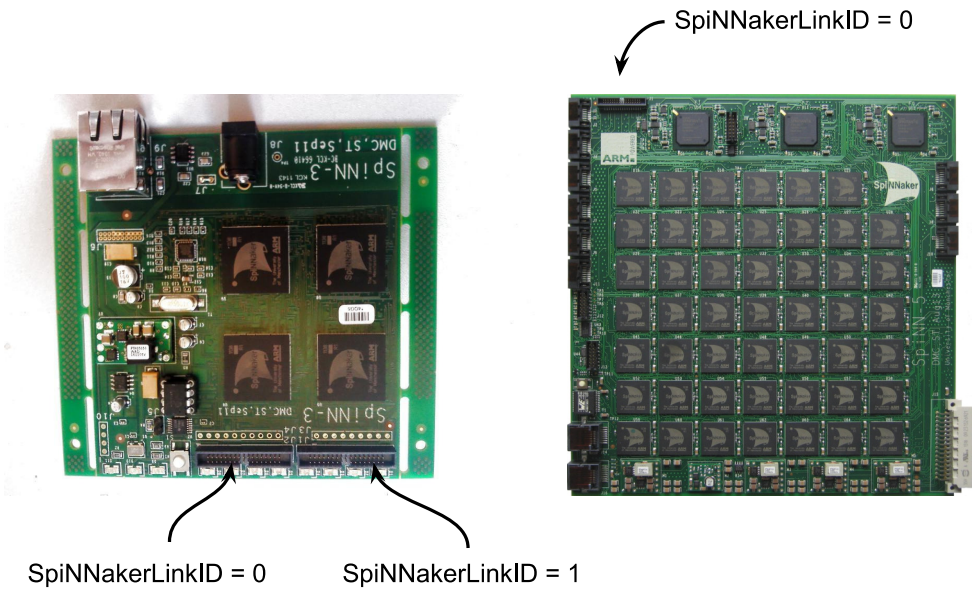
Using an external device: Calls from PYNN

```
import pynn.spinnaker as p
p.setup( timestep=1.0,
         min_delay = 1.0,
         max_delay = 32.0)

# set up populations
pop =  p.Population(
     1, p.IfCurExp, {} label='pop1'))


# set populations to record spikes
pop.record()

# run the simulation for 10000 ms
p.run(10000)
```

Using an external device: Calls from PYNN

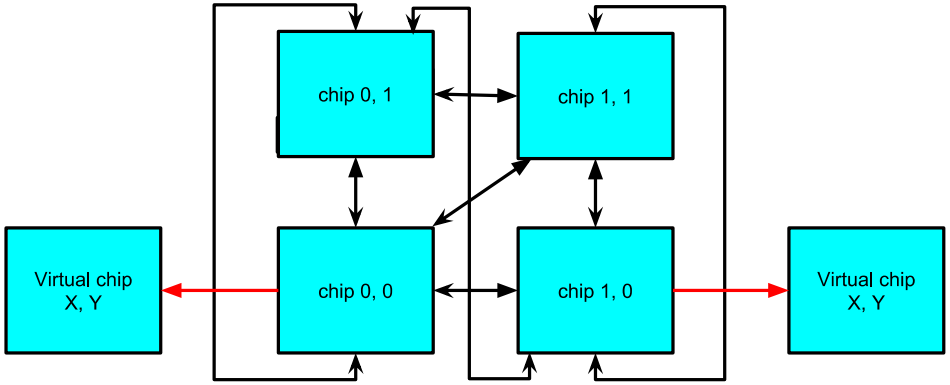
```
import pynn.spinnaker as p
p.setup( timestep=1.0,
         min_delay = 1.0,
         max_delay = 32.0)

# set up populations
pop =  p.Population(
     1, p.ExternalDevice, {
          'spinnaker_link':0,
          'board_address':None OR 192.168.0.253,} label='pop1'))

# set populations to record spikes
pop.record()      External devices cant be recorded

# run the simulation for 10000 ms
p.run(10000)
```

Which SpiNNaker Link is which?



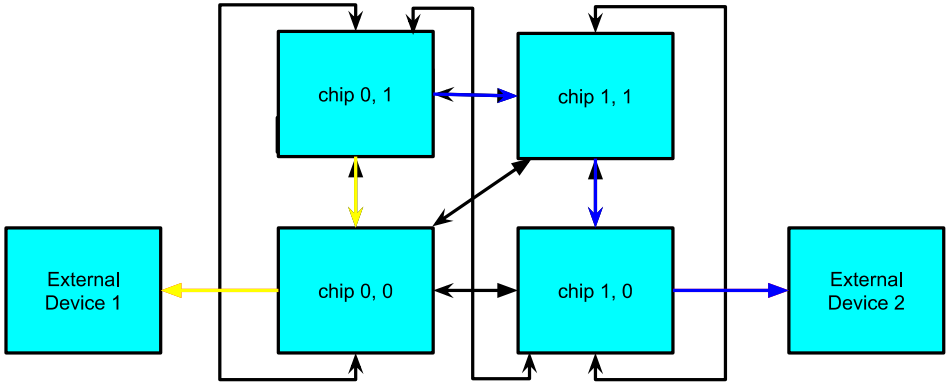
How this works in detail

1. Every Spinnaker link is defined as a link to a virtual chip
2. Your device vertex is then placed within this virtual chip.



Why does it matter?

1. Routing won't work if mixed up



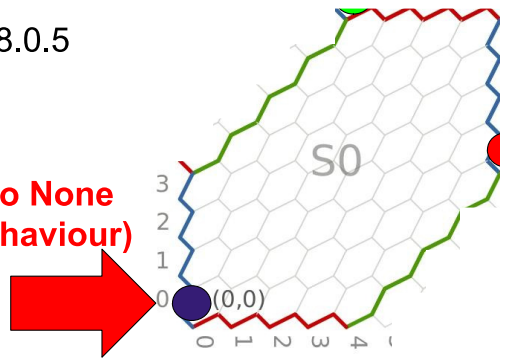
Board Address?

192.168.0.1

192.168.0.3

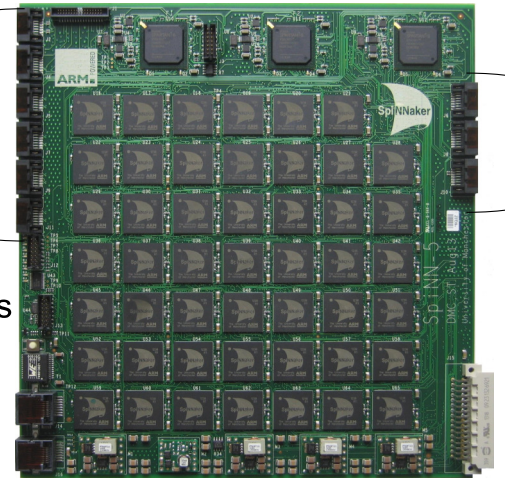
192.168.0.5

When set to None (default behaviour)



SATA Link connected devices!

torus SATA links



SATA links

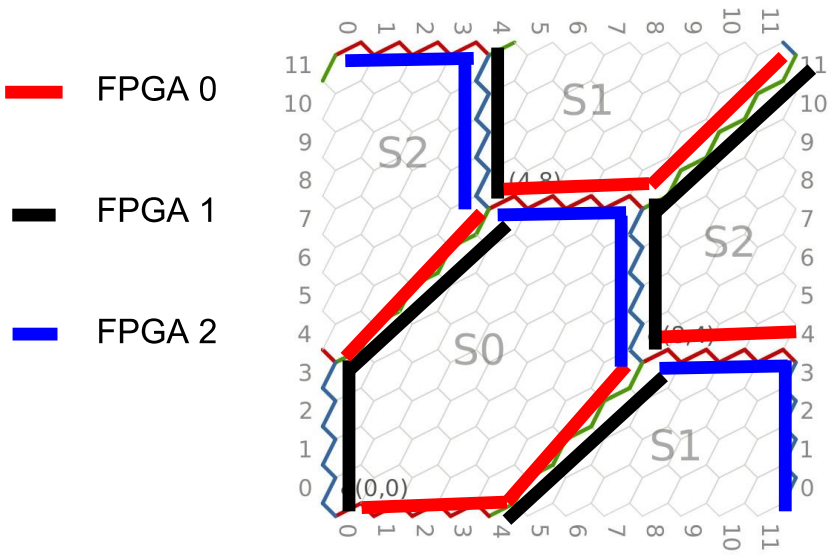
People who have devices

1. Bernabe Linares-Barranco
2. Jorg Conradt

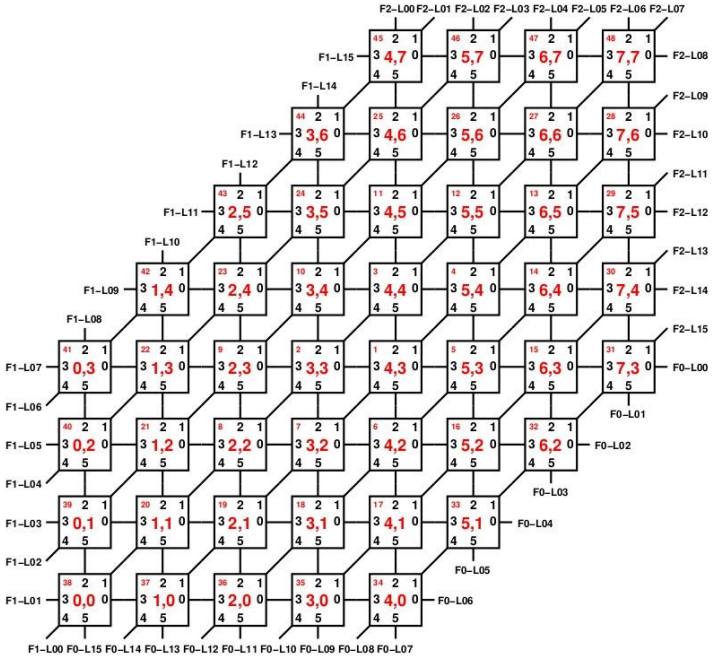
How to represent this in your PyNN scripts.

```
p.Population(2000, external_devices.ArbitraryFPGADevice,
                {
                    'fpga_link_id': 12,
                    'fpga_id': 1,
                    'board_address': None OR 192.168.0.1
                },
               label='External sata thing')
```

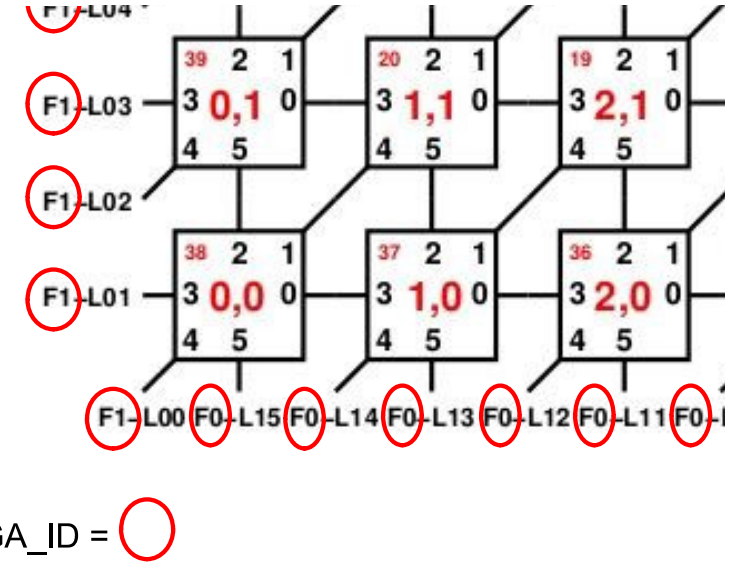

How do FPGA's work in Multi-board machines?



What the input parameters mean?

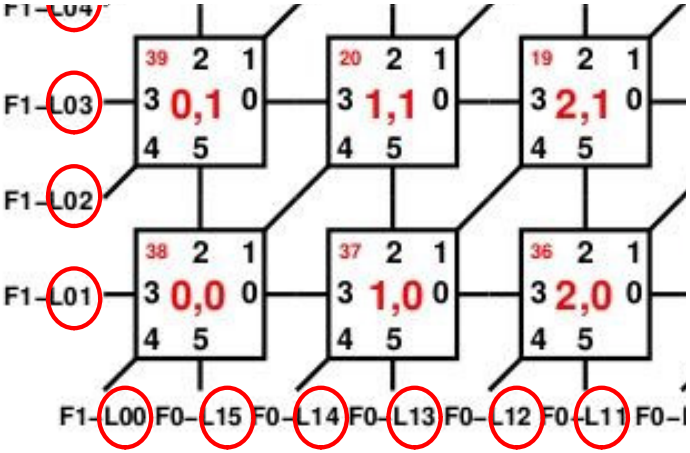


What the input parameters mean?



FPGA_ID = ◯

What the input parameters mean?



FPGA__link_ID =

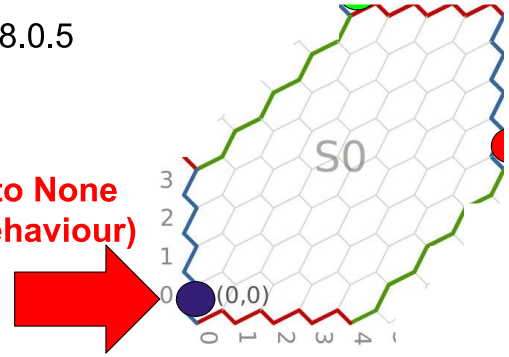
Board Address again

192.168.0.1

192.168.0.3

192.168.0.5

When set to None (default behaviour)



What you need to do to get SATA links working for your device.

1. Reprogram the FPGA's to support the communication between device and PyNN related models.
2. **The reprogramming needs to result in a disconnected edge between two chips who's communication is done through the FPGA.**
3. Extend or use the ArbitaryFPGADevice vertex to represent any extra constraints you need.

Summary

1. Discussed External devices plugged in through the SpiNNaker Link.
2. Discussed External devices plugged in through the FPGA / SATA connector.
3. Discussed How the FPGA's interact in the communication fabric.

6th SpiNNaker Workshop

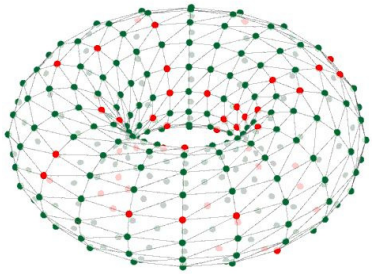
Day 4

September 8th 2016

Time	Session	Presenter
09:00	Adding new models of synaptic plasticity	JK
09:45	Graph Front End – further details	ABS (AGR)
10:30	Coffee	
11:00	Lab time	
12:00	Lunch	
13:00	Using big SpiNNaker machines remotely: The HBP portal	AGR
13:30	Lab time (coffee at 15:00)	
15:30	Demonstration of NENGO language and environment	TBC
16:30	Close	

Manchester, UK

Adding new models of synaptic plasticity



Jamie Knight

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

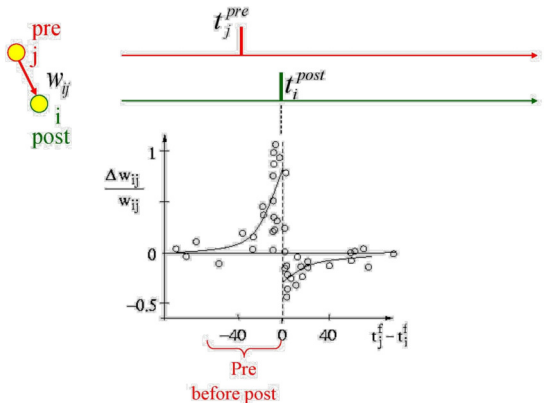
SpiNNaker

Outline

- Introduction to spike-timing dependent plasticity
- Simulating STDP
- Limitations of pair-based STDP
- Triplet STDP
- SpiNNaker implementation

Introduction to spike-timing dependent plasticity

"Cells that fire together, wire together"



http://www.scholarpedia.org/article/Spike-timing_dependent_plasticity

Simulating STDP - Traces

Pre-synaptic trace

$$\frac{dx_j}{dt} = -\frac{x_j}{\tau_x} + \sum_{t_j^f} \delta(t - t_j^f)$$

Post-synaptic trace

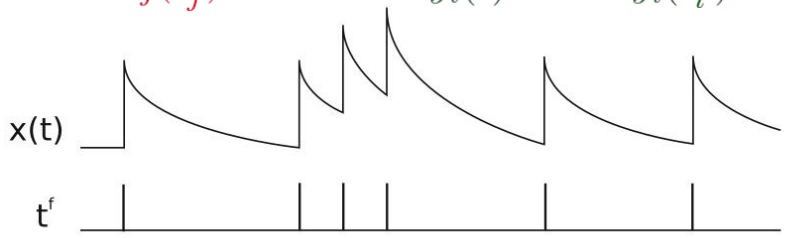
$$\frac{dy_i}{dt} = -\frac{y_i}{\tau_y} + \sum_{t_i^f} \delta(t - t_i^f)$$

At pre-synaptic spike time

$$x_j(t) = 1 + x_j(t_j^f) e^{-\frac{t - t_j^f}{\tau_x}}$$

At post-synaptic spike time

$$y_i(t) = 1 + y_i(t_i^f) e^{-\frac{t - t_i^f}{\tau_y}}$$



Morrison, A., Diesmann, M., & Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. Biological Cybernetics, 98, 459–478.

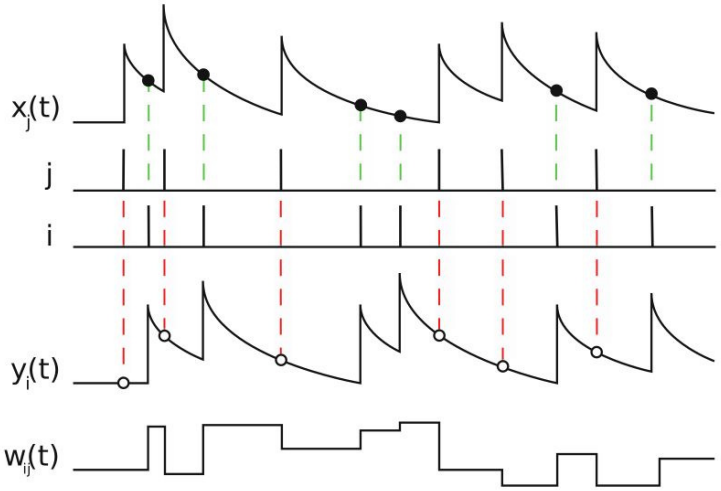
Simulating STDP - Weight update

Pre-synaptic weight update

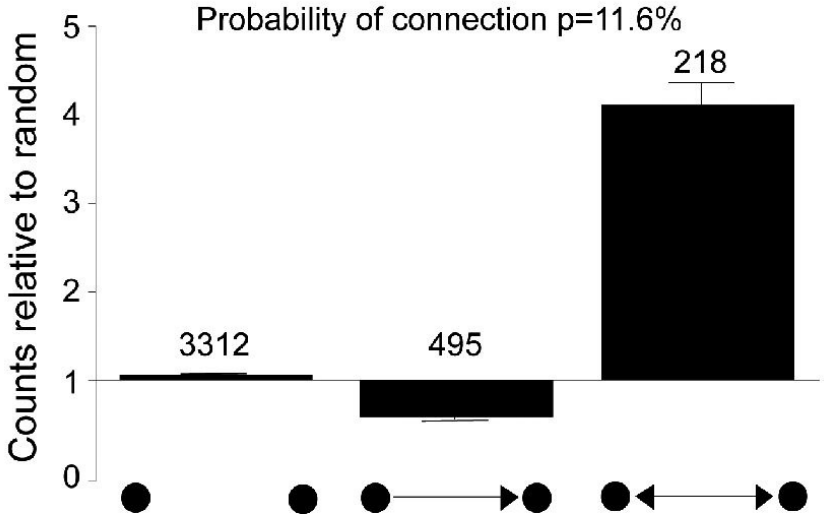
$$\Delta w_{ij}^{-} = F_{-}(w_{ij}) y_i(t_j^f)$$

Post-synaptic weight update

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f)$$

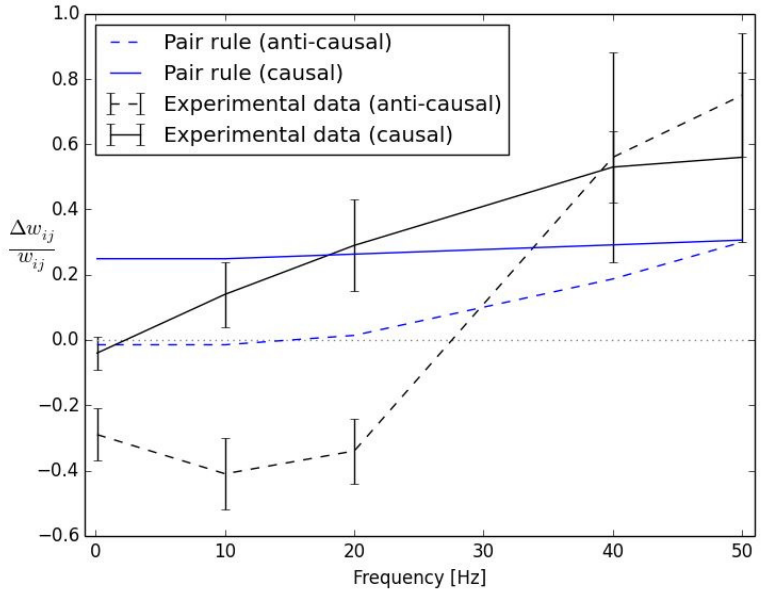


Limitations of pair-based STDP



Song, S., Sjöström, P. J., Reigl, M., Nelson, S., & Chklovskii, D. B. (2005). Highly nonrandom features of synaptic connectivity in local cortical circuits. PLoS Biology, 3(3), 0507–0519.

Limitations of pair-based STDP



Sjöström, P. J., Turrigiano, G. G., & Nelson, S. B. (2001). Rate, timing, and cooperativity jointly determine cortical synaptic plasticity. *Neuron*, *32*(6), 1149–64.

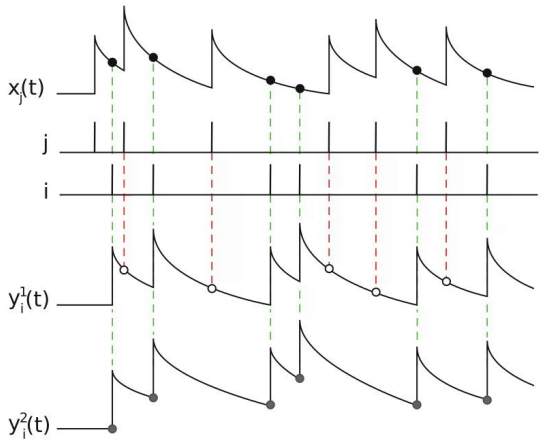
Triplet STDP

Slow post-synaptic trace

$$y_i^2(t) = \left(1 + y_i^2(t_i^f)\right) e^{-\frac{t - t_i^f}{\tau_y^2}}$$

Post-synaptic weight update

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f) y_i^2(t_i^{f-})$$



Pfister, J. P., & Gerstner, W. (2006). Triplets of spikes in a model of spike timing-dependent plasticity. The Journal of Neuroscience : The Official Journal of the Society for Neuroscience, 26(38), 9673–82.

SpiNNaker - Pair traces

timing_pair_impl.h
line 7

```
typedef int16_t post_trace_t;
```

timing_pair_impl.h
lines 46-49

```
static inline post_trace_t timing_get_initial_post_trace()
{
  return 0;
}
```


SpiNNaker - Triplet traces

timing_triplet_impl.h
line 7

```
typedef struct post_trace_t
{
  int16_t y1;
  int16_t y2;
} post_trace_t;
```

timing_triplet_impl.h
lines 46-49

```
static inline post_trace_t timing_get_initial_post_trace()
{
  return (post_trace_t){.y1 = 0, .y2 = 0};
}
```


SpiNNaker - Pair trace update

timing_pair_impl.h
lines 54-66

$$y_i(t) = 1 + y_i(t_i^f)e^{-\frac{t-t_i^f}{\tau_y}}$$

```
// Get time since last spike
uint32_t delta_time = time - last_time;

// Decay previous trace (y)
int32_t new_y = STDP_FIXED_MUL_16X16(last_trace,
  DECAY_TAU_Y(delta_time));

// Add energy caused by new spike to trace
new_y += STDP_FIXED_POINT_ONE;

log_debug("\tdelta_time=%d, y=%d\n", delta_time, new_y);

// Return new trace_value
return (post_trace_t)new_y;
```


SpiNNaker - Triplet trace update

timing_triplet_impl.h
lines 77-87

$$y_i^2(t) = \left(1 + y_i^2(t_i^f)\right)e^{-\frac{t-t_i^f}{\tau_y^2}}$$

```
// Y2 is sampled in timing_apply_post_spike BEFORE the spike
// Therefore, if this is the first spike, y2 must be zero
int32_t new_y2;
if(last_time == 0)
{
  new_y2 = 0;
}
// Otherwise, add energy of spike to last value and decay
else
{
  new_y2 = STDP_FIXED_MUL_16X16(
    last_trace.y2 + STDP_FIXED_POINT_ONE,
    DECAY_TAU_Y2(delta_time));
}
```

SpiNNaker - Pair weight update

timing_pair_impl.h
lines 136-150

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f)$$

```
uint32_t delta_t = time - last_pre_time;

// If spikes are not co-incident
if (delta_t > 0)
{
  // Calculate x(time) = x(last_pre_time) * e^(-delta_t/tau_x)
  int32_t x = STDP_FIXED_MUL_16X16(last_pre_trace,
    DECAY_TAU_X(delta_t));

  log_debug("\t\t\tdelta_t=%u, x=%d\n",
    delta_t, x);

  // Apply potentiation to synapse state
  return weight_one_term_apply_potentiation(previous_state, x);
}
```

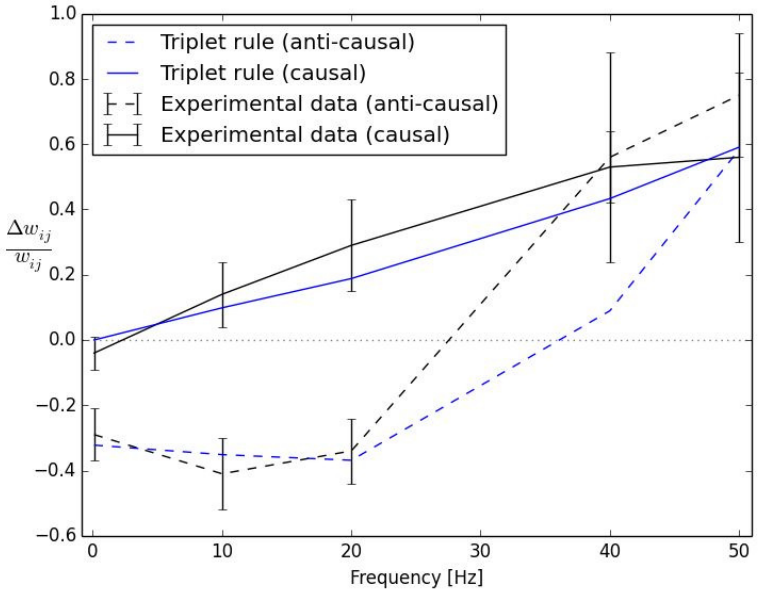

SpiNNaker - Triplet weight update

timing_triplet_impl.h
lines 165-179

$$\Delta w_{ij}^{+} = F_{+}(w_{ij}) x_j(t_i^f) y_i^2(t_i^{f-})$$

```
if (delta_t > 0)
{
  // Calculate x(time) = x(last_pre_time) * e^(-delta_t/tau_x)
  int32_t x = STDP_FIXED_MUL_16X16(last_pre_trace,
    DECAY_TAU_X(delta_t));

  // Multiply this by y2(time) to get triplet term
  int32_t x_y2 = STDP_FIXED_MUL_16X16(x, trace.y2);

  log_debug("\t\t\tdelta_t=%u, x=%d, y2=%d, x_y2=%d\n",
    delta_t, x, trace.y2, x_y2);

  // Apply potentiation to synapse state
  return weight_one_term_apply_potentiation(previous_state, x_y2);
}
```


Triplet model

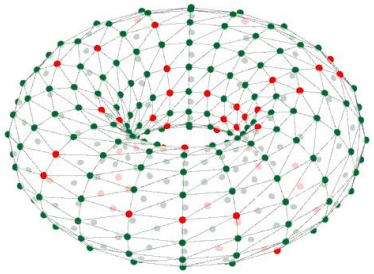


Thank you!

Any questions?

james.knight@manchester.ac.uk

Graph Front End - Advanced



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

EPSRC

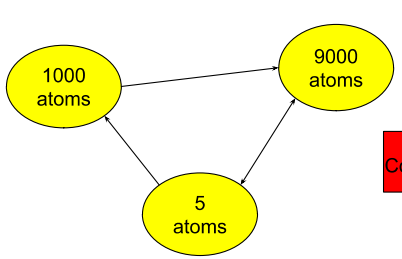
SpiNNaker

Contents

- Working with application graphs
- Buffered recordings
- Auto pause and resume
- Provenance data

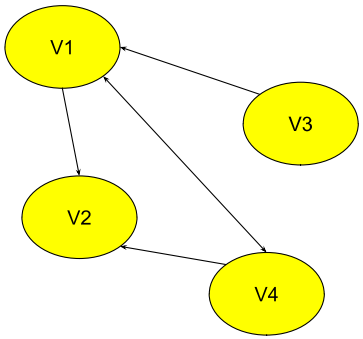
Supported graphs (PACMAN)

Application Graph



Converts into

Machine Graph



Needs breaking down into core sized chunks

Already has a 1:1 ratio between vertices and core.

Basic script to add application vertices into the graph

```
import spinnaker_graph_front_end as front_end

from spinnaker_graph_front_end.examples.Conways.conways_application_cell\
    import ConwayApplicationCell

# set up the front end and ask for the detected machines dimensions
front_end.setup()

front_end.add_application_vertex_instance(
      ConwayApplicationCell(800, "ConwayCells"))

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```

Creating a new type of application vertex

```
from pacman.model.graphs.application.impl.application_vertex  import ApplicationVertex
from pacman.model.resources.resource_container import ResourceContainer
from pacman.model.resources.cpu_cycles_per_tick_resource import CPUCyclesPerTickResource
from pacman.model.resources.dtcm_resource import DTCMResource
from pacman.model.resources.sdram_resource import SDRAMResource

class ConwayApplicationCell(ApplicationVertex):
    """ Represents a collection of cells within the 2D grid
    """
    def __init__(self, n_atoms, label):
        ApplicationVertex.__init__(self, label=label, max_atoms_per_core=200)
        self._n_atoms = n_atoms

    def get_resources_used_by_atoms(self, vertex_slice):
        resources = ResourceContainer(
            sdram=SDRAMResource(4 * vertex_slice.n_atoms),
            dtcm=DTCMResource(4 * vertex_slice.n_atoms),
            cpu_cycles=CPUCyclesPerTickResource(100 * vertex_slice.n_atoms)
        )
```


Creating a new type of application vertex

```
    def create_machine_vertex(
            self, vertex_slice, resources_required, label=None, constraints=None):

        # return a partitioned vertex that's designed to handle multiple atoms within it
        return ConwayMachineCell(
            label=label, resources_required=resources_required,
            constraints=constraints)

    @property
    def n_atoms(self):

        # return the atoms this vertex contains
        return self._n_atoms
```


Basic Script adding application edges

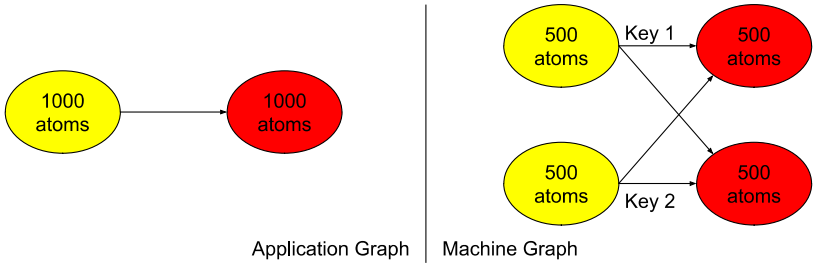
```
import spinnaker_graph_front_end as front_end

# build and add application vertex
vertex = ConwayApplicationCell(800, "ConwayCells")
front_end.add_application_vertex_instance(vertex)

# build an application edge
front_end.add_application_edge_instance(
    ApplicationEdge(vertex, vertex), "State")

front_end.run(5000)

front_end.stop()
```

Partition id

Data generation

```
...
def generate_application_data_specification(
        self,  spec, placement, graph_mapper, application_graph, machine_graph,
        routing_info, iptags,  reverse_iptags, machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
    region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
    region=1, size=8, label="inputs")
...

# get slice of atoms for machine vertex
vertex_slice = graph_mapper.get_slice(placement.vertex)
```

Application vertex c code



Hints:

1. You need to be able to distinguish from the received key which atoms it effects on the core you are writing the data for
2. You need to execute your application c code for every atom on the core

Buffered Recordings

Problem

1. SDRAM is limited on the SpiNNaker machines.
2. Recording of data is more reliable on SDRAM than live transmissions.
3. Simulations run for long periods of time gathering data.

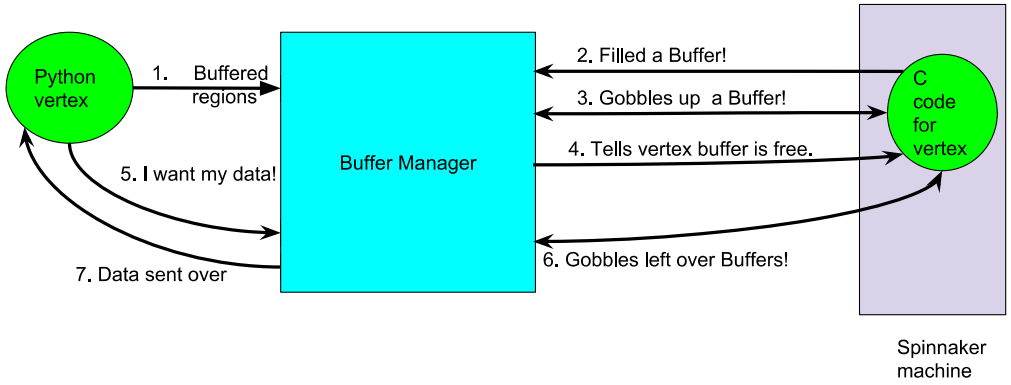
Buffered Recordings

Solution

1. Store data in small chunks called buffers
2. During simulation, or during a pause, extract the buffers

NOTE: This only works in tandem with the simulation.h and data_specification.h and python interfaces.

How does a extracted buffered data region work?



Buffered Recording - Python

```
class MyBufferedVertex(..., ReceiveBuffersToHostBasicImpl):

def __init__(...):
    ReceiveBuffersToHostBasicImpl.__init__(self)
    ...
```


Buffered Recording - Python

```
class MyBufferedVertex(..., ReceiveBuffersToHostBasicImpl):
    ...

    def generate_data_spec(...):
        ...
        spec.reserve_memory_region(
            region=2, size=self.get_recording_data_size(3), label="recording")
        ...
        spec.reserve_memory_region(
            region=6, size=self.get_buffer_state_region_size(3), label="state")
        ...
        ...
        self.reserve_buffer_regions(spec, 6, [4,5,7], [1000000, 1000000, 100000])
        ...
        spec.switch_write_focus(2)
        self.write_recording_data(spec, iptags, [1000000, 1000000, 100000], 16384)
        ...
```

Number of buffered regions

Extra region for storing buffered state

Buffered region ids

Allocated buffer sizes

IP Tags holder

Buffer size before request sent

Buffered Recording - C

```
static uint32_t recording_flags = 0;

void c_main(void) {
    ...
    address_t address = data_specification_get_data_address();
    address_t recording_region = data_specification_get_region(2, address);
    uint8_t *regions_to_record[] = {4,5,7};

    bool success = recording_initialize(
        3, regions_to_record, recording_region, 6, &recording_flags);

    ...
    simulation_run();
}
```

Buffered region ids (channels 0, 1 and 2)

Number of buffered regions

Extra region for storing buffered state

Buffered Recording - C

```
...
void timer_callback(uint unused0, uint unused1) {
    ...
    if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
        recording_finalize();
        ...
    }

if (recording_is_channel_enabled(recording_flags, 0)) {

        uint32_t data = 23;
        recording_record(0, &data, 4);
    }

    recording_do_timestep_update(time);
    ...
}
```

Recording channel number (= region 4)

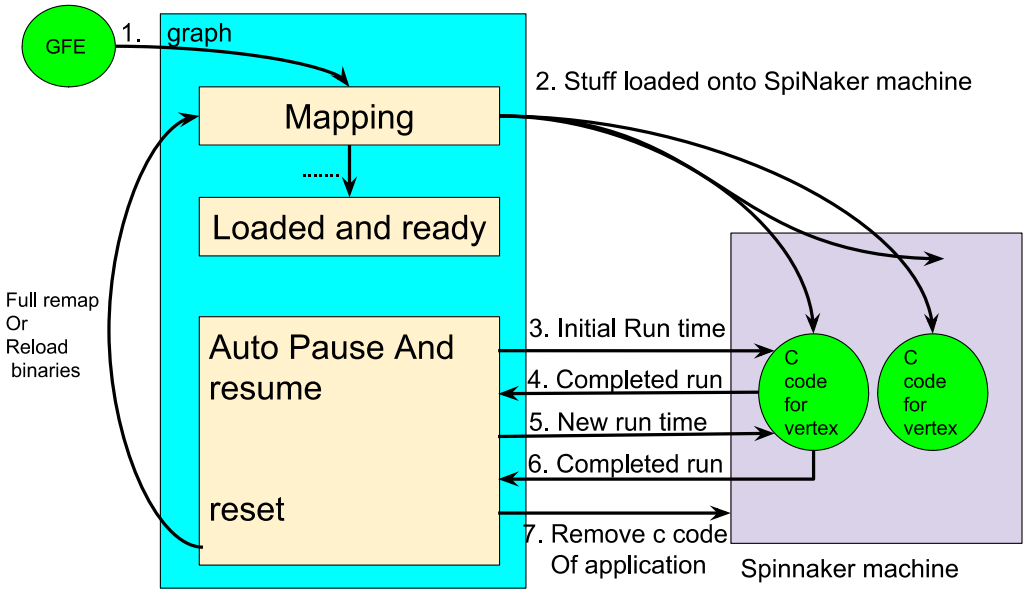
Pointer to data to record

Size of data to record in bytes

Auto pause and resume functionality

1. Provides the ability to run a simulation for multiple periods without remapping the application.
2. Provides the ability to extract buffers without affecting the running simulation.
3. Supports the ability to reset a simulation to the state at t=0.

How Auto Pause and Resume works.



Auto Pause and Resume - Python

```
class AbstractPopulationVertex(..., AbstractChangableAfterRun):
...
def __init__(.....):
    AbstractChangableAfterRun.__init__(self)

    # bool for if state has changed.
    self._change_requires_mapping = True

@property
def requires_mapping(self):
    # determine if there are changes within which require a remapping
    return self._change_requires_mapping

def mark_no_changes(self):
    # restart the tracking of changes
    self._change_requires_mapping = False

def set_recording_spikes(self):
    self._change_requires_mapping = not self._spike_recorder.record
    self._spike_recorder.record = True
```


Auto Pause and Resume - C

```
...
void timer_callback(uint unused0, uint unused1) {
    ...

    if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
        simulation_exit();
        simulation_handle_pause_resume(resume_callback);
        ...
    }
}

void resume_callback() {

    // restart the recording just before resuming
    if (!initialise_recording()) {
        rt_error(RTE_SWERR);
    }
}
```

Provenance data gatherers

1. Data that can be used to prove 2 simulations are equivalent to each other.
2. Data that can also be used for debug purposes.
3. Is stored in XML and searched through for errors by the main tools.
4. Every vertex can provide its own provenance data.

Provenance example output

```
<provenance_data_items name="my_object">
  <provenance_data_items name="my_category">
    <provenance_data_item name="my_item">0</provenance_data_item>
    <provenance_data_item name="my_other_item">0</provenance_data_item>
  </provenance_data_items>
</provenance_data_items>

<provenance_data_items name="0_0_5_my_vertex">
  <provenance_data_items name="my_category">
    <provenance_data_item name="my_machine_value">0</provenance_data_item>
  </provenance_data_items>
</provenance_data_items>
```


Local Provenance Data - Python

```
class MyVertex(..., AbstractProvidesLocalProvenanceData):
    ...

    def get_local_provenance_data(self)
        self._data_items = list()

        # store data in a provenance data item
        self._data_items.append(
            ProvenanceDataItem(
                ["my_object" "my_category", "my_item"], my_value))
        self._data_items.append(
            ProvenanceDataItem(
                ["my_object", "my_category", "my_other_item"], my_other_value,
                report=(my_other_value > error_value),
                message="value {} was bigger than expected ({})".format(
                    my_value, error_value))
        ...

        # return provenance items
        return self._data_items
```

Hierarchy of categories and names used to group items in XML

debug arguments

Simulation Provenance Data - Python

```
class MyVertex(..., ProvidesProvenanceDataFromMachineImpl):
    ...
    def get_provenance_data_from_machine(self, transceiver, placement):
        provenance_data = self._read_provenance_data(transceiver, placement)

        # translate system specific provenance data items
        provenance_items = self._read_basic_provenance_items(
            provenance_data, placement)

        # translate application specific provenance data items
        provenance_data = self._get_remaining_provenance_data_items(
                provenance_data)
        my_value = provenance_data[0]
        label, x, y, p, names = self._get_placement_details(placement)

        # translate into provenance data items
        provenance_items.append(
            ProvenanceDataItem(
                self._add_names(names, ["my_category", "my_machine_value"]),
                my_value))

        return provenance_items
```

Simulation Provenance Data - Python

```
class MyVertex(..., ProvidesProvenanceDataFromMachineImpl):
    ...

    def __init__(self, …)
        ProvidesProveanceDataFromMachineImpl.__init__(self, 9, 1)
        ...

    def generate_data_spec(...):
        ...
        self.reserve_provenance_data_region(spec)
```

Provenance Region

Number of custom provenance data items

Simulation Provenance Data - C

```
static my_value = 0;

void c_main(void) {

    ...
    if (!simulation_initialise(
            system_region, APPLICATION_NAME_HASH,
            &timer_period, &simulation_ticks,
            &infinite_run, SDP,
            get_provenance_data,
            data_specification_get_region(9, address))) {
        log_error("Error in initialisation - exiting!");
        rt_error(RTE_SWERR);
    }
    …
}

void get_provenance_data(address_t provenance_data_address) {
    provenance_data_address[0] = my_value;
}
```

Provenance Region

Summary

1. Application graphs
2. Buffered recording
3. Auto pause and resume
4. Provenance data gathering

6th SpiNNaker Workshop

Day 5

September 9th 2016

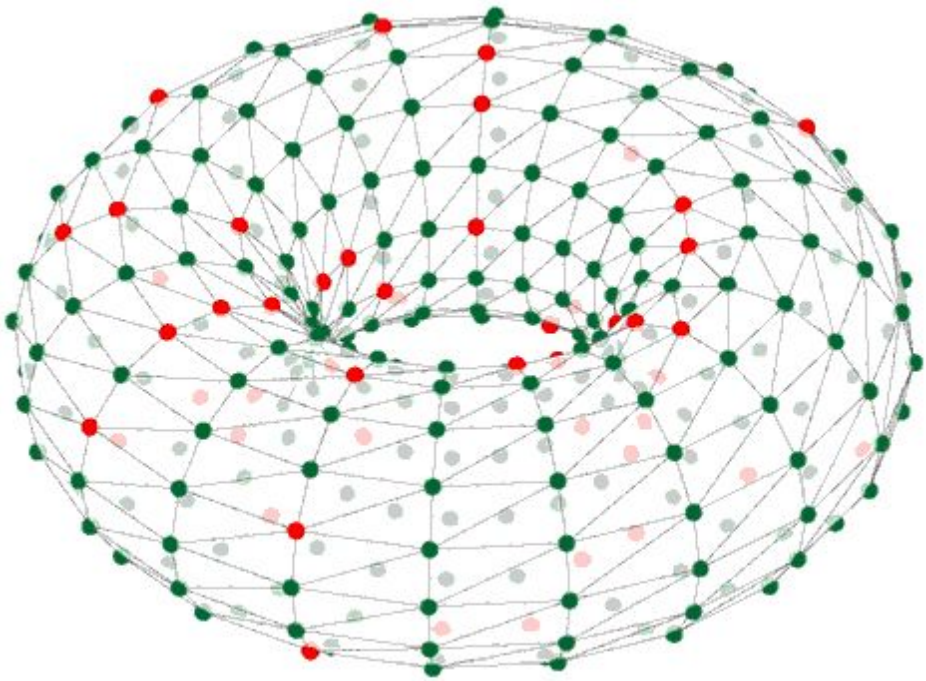
Time	Session	Presenter
09:00	Lab time	
10:30	Coffee	
11:00	Lab time	
12:00	Lunch and close	

Manchester, UK

6th SpiNNaker Workshop

Lab Manuals

September
5th- 9th 2016



Manchester, UK

SpiNNaker

Intro Lab

This lab is meant to expose workshop participants to examples of problems which can be applied to the SpiNNaker architecture.

Installation

The software installation instructions can be found here:
https://spinnakermanchester.github.io/latest/spynnaker_install.html
https://spinnakermanchester.github.io/latest/gfe_install.html

File download

All of these examples can be found here:
https://spinnakermanchester.github.io/latest/intro_lab.html

Please download and open a terminal at the top level of the folder.

Run Applications

Below is a list of applications with the corresponding folders and execution commands, please run each script as it currently stands, and attempt to understand what the application is doing.

1. Neural Network Synfire Chain
2. Conductive Material with Applied Heat
3. Sudoku Game Through Neural Network
4. Graphic Ray Tracer of an Environment
5. Simple Learning Network

Neural Network Synfire Chain

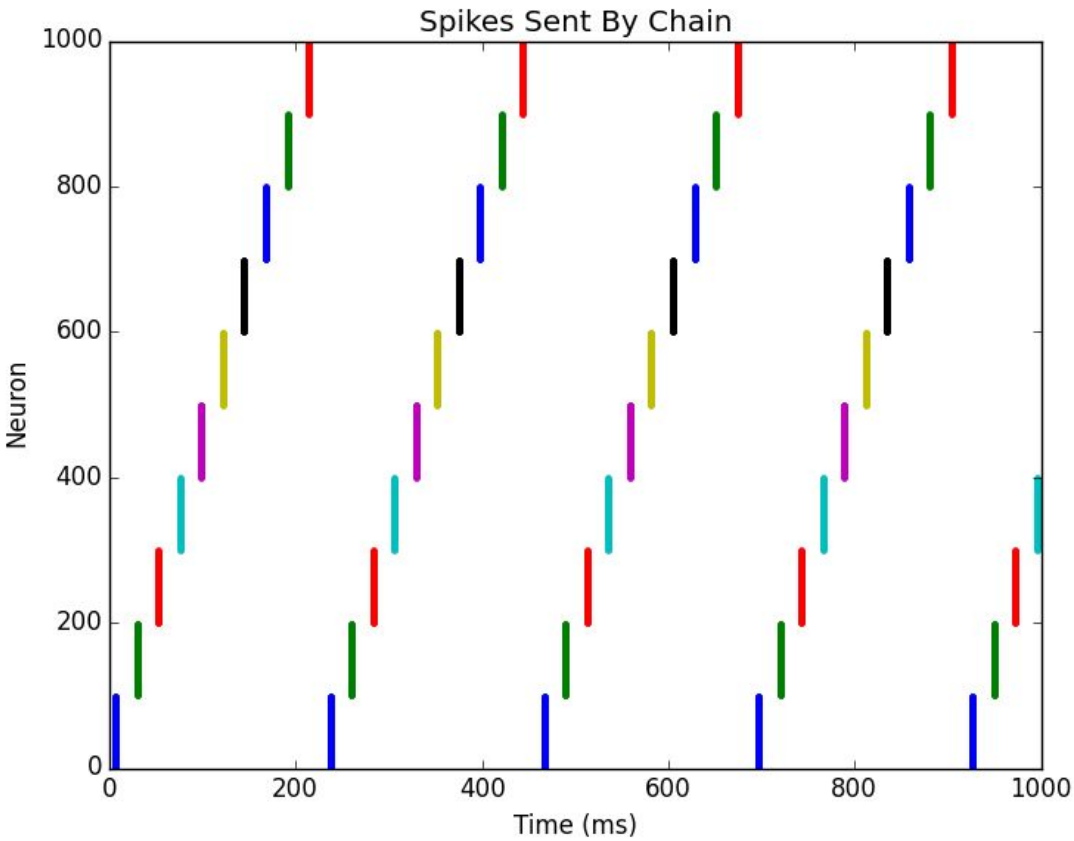


Figure 1: The output from a simple Synfire chain.

To run this example, from the top level of the folder type:

```
cd synfire
python synfire.py
```

A plot like the above should appear.

This example shows a PyNN Neural Network with a chain of 10 populations of 100 neurons each, where 10 neurons from each population excite all the neurons in the next population in the chain. The first population is then stimulated at the start of the simulation to start the chain running.

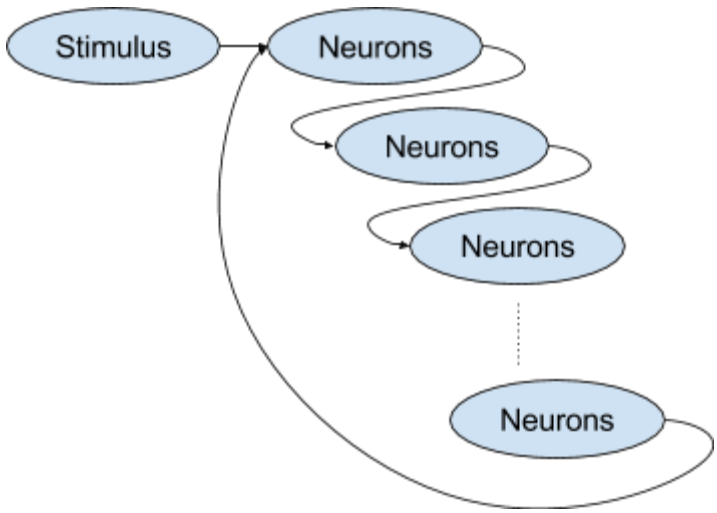


Figure 2: The Synfire Chain of Populations

Conductive Material with Applied Heat

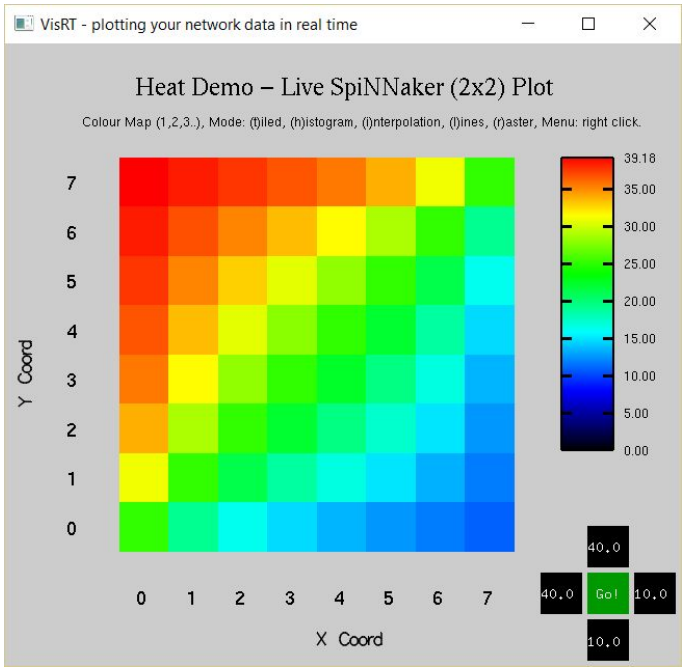


Figure 3: The output from the conductive material simulation

To run this example, from the top level of the folder type:

```
cd heat_demo
python heat_demo.py
```

A visualiser should appear here, as shown in Figure 3. You can press “9” to randomize the heat applied at each edge of the simulation, or press select a black square and press “+” to increase the temperature, or “-” to decrease it, followed by “g” to update it.

This example shows a piece of conductive material (e.g. a metal sheet) which is represented by a collections of cells which represent atoms of the material. Temperature is transferred between the atoms of the material in the simulation by sending packets over the SpiNNaker network. Figure 4 shows this application in graphical form.

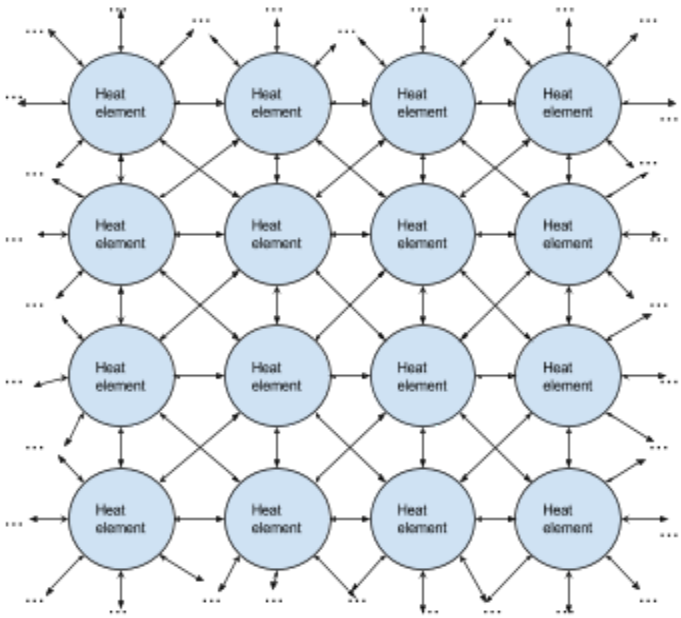


Figure 4: The conductive material application in graphical form

Sudoku Game Through Neural Network

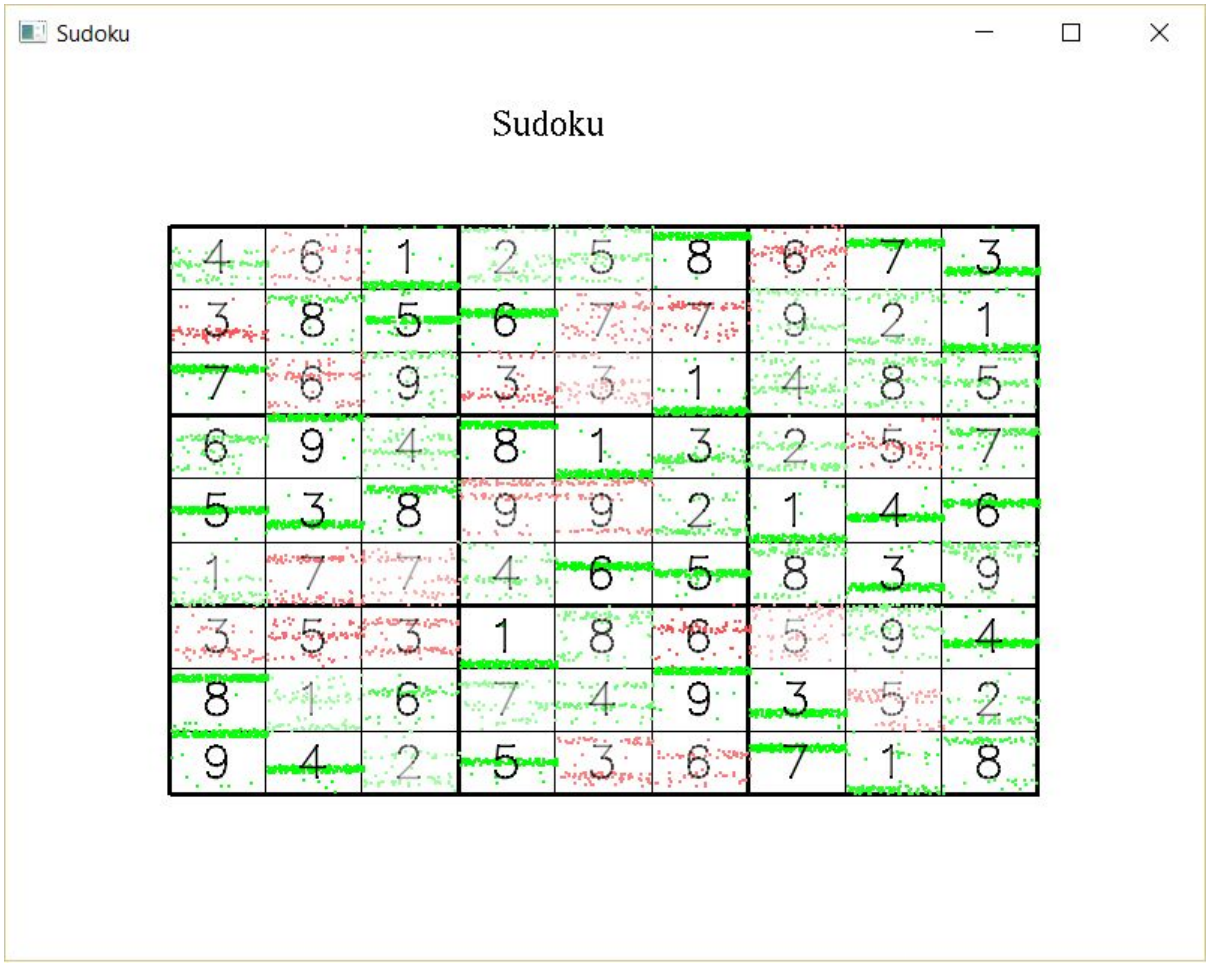


Figure 5: The output from the Sudoku game application

To run this example, from the top level of the folder type:

```
cd sudoku
python sudoku.py
```

A visualiser will pop up, which is shown in Figure 5.

This example shows a PyNN neural network which describes a neural network for running sudoku problems. The spikes representing each cell are shown behind each number, with green output indicating that the value is valid according to the rules of Sudoku and red output indicating that the value is invalid. The problem to be solved is described near the top of the sudoku.py file, with 0s representing values to be computed. Note that on a small SpiNNaker board, the network is not always successful at solving the problem.

Graphic Ray Tracer of an Environment

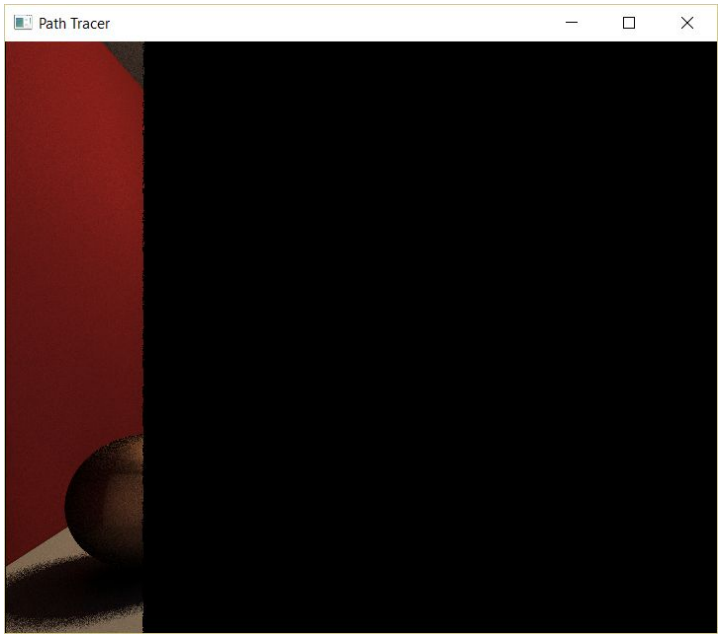


Figure 6: The output from the graphical ray tracer application

To run this example, from the top level of the folder type:

```
cd ray_trace
python ray_trace.py
```

A visualiser will pop up, which is shown in Figure 6.

This example shows a ray tracing application on SpiNNaker. This has been designed so that to operate in parallel; the more cores in use, the faster it completes. Note that it is still quite slow, and you may have to click on the window to force it to update.

Simple Learning Network

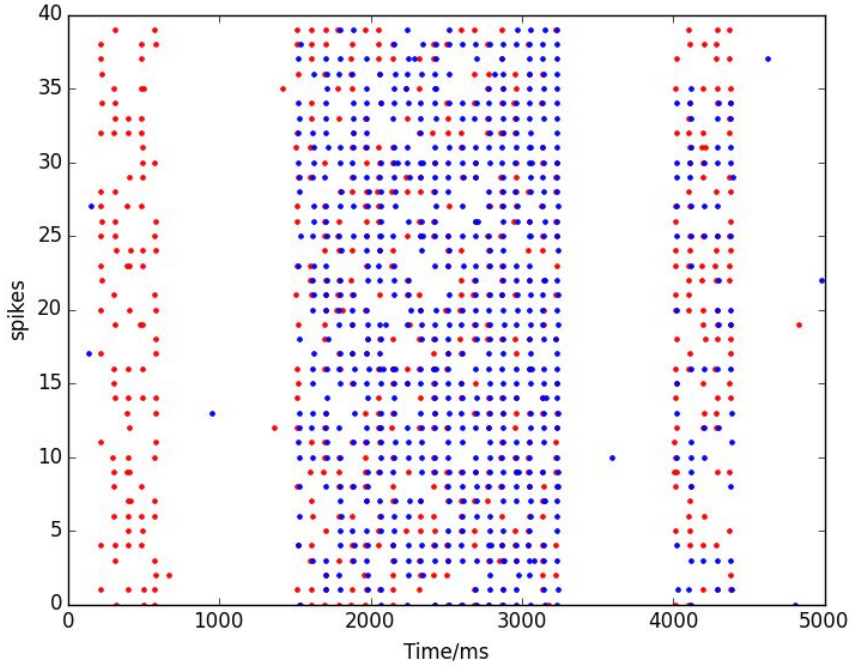


Figure 7: The output from the learning application

To run this example, from the top level folder type:

```
cd learning
python learning.py
```

A plot like the above should appear.

This example shows the spike outputs from two populations of neurons. At the start, only one of the populations spikes regularly. In the middle, some learning is done, and at the end, both populations spike regularly.

Running PyNN Simulations on SpiNNaker

Introduction

This manual will introduce you to the basics of using the PyNN neural network language on SpiNNaker neuromorphic hardware.

Installation

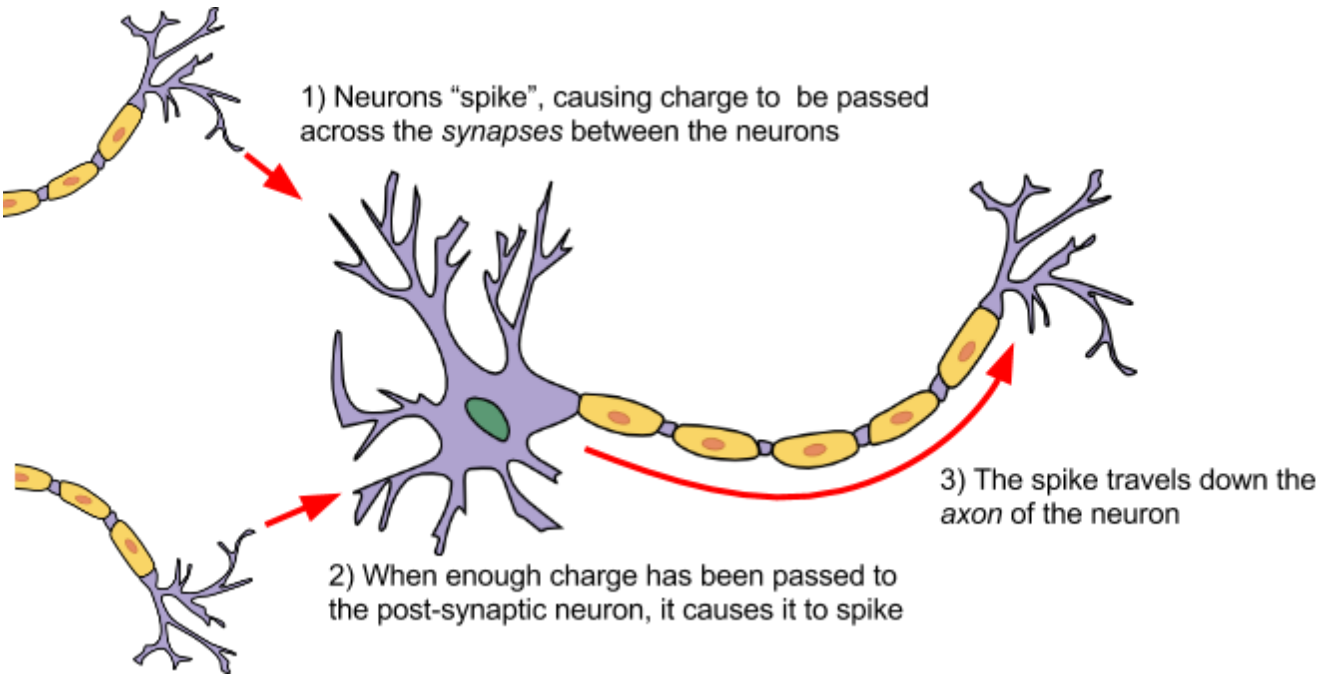
The PyNN toolchain for SpiNNaker (sPyNNaker), can be installed by following the instructions available from here:

http://spinnakermanchester.github.io/2015.005.Arbitrary/PyNNOnSpinnakerInstall.html

Matplotlib is marked as optional, but you will also need to install this dependency to complete some of the forthcoming exercises.

Spiking Neural Networks

Biological neurons have been observed to produce sudden and short increases in voltage, commonly referred to as spikes. The spike causes a charge to be transferred across the synapse between neurons. The charge from all the presynaptic neurons connected to a postsynaptic neuron builds up, until that neuron releases the charge itself in the form of a spike. The spike travels down the axon of the neuron which then arrives after some delay at the synapses of that neuron, causing charge to be passed forward to the next neuron, where the process repeats.



Artificial spiking neural networks tend to model the membrane voltage of the neuron in response to the incoming charge over time. The voltage is described using a differential equation over time, and the solution to this equation is usually computed at fixed time-steps within the simulation. In addition to this, the charge or current flowing across the synapse can also be modelled over time, depending on the model in use.

The charge can result in either an excitatory response, in which the membrane voltage of the postsynaptic neuron increases or an inhibitory response, in which the membrane voltage of the postsynaptic neuron decreases as a result of the spike.

The PyNN Neural Network Description Language

PyNN is a language for building neural network models. PyNN models can then be run on a number of simulators without modification (or with only minor modifications), including SpiNNaker. The basic steps of building a PyNN network are as follows:

1. Setup the simulator
2. Create the neural *populations*
3. Create the *projections* between the populations
4. Setup data recording
5. Run the simulation
6. Retrieve and process the recorded data

An example of this is as follows:

```
import pyNN.spiNNaker as p
p.setup(timestep=1.0)
pop_1 = p.Population(1, p.IF_curr_exp, {}, label="pop_1")
input = p.Population(1, p.SpikeSourceArray,
        {'spike_times': [[0]]}, label="input")
input_proj = p.Projection(input, pop_1, p.OneToOneConnector(
    weights=5.0, delays=1), target="excitatory")
pop_1.record()
pop_1.record_v()
p.run(10)

import pylab
time = [i[1] for i in v if i[0] == 0]
membrane_voltage = [i[2] for i in v if i[0] == 0]
pylab.plot(time, membrane_voltage)
pylab.xlabel("Time (ms)")
pylab.ylabel("Membrane Voltage")
pylab.axis([0, 10, -75, -45])
pylab.show()

spike_time = [i[1] for i in spikes]
spike_id = [i[0] for i in spikes]
pylab.plot(spike_time, spike_id, ".")
pylab.xlabel("Time (ms)")
pylab.ylabel("Neuron ID")
pylab.axis([0, 10, -1, 1])
pylab.show()
```

This example runs using a 1.0ms timestep. It creates a single input source (A *SpikeSourceArray*) sending a single spike at time 0, connected to a single neuron (with model *IF_curr_exp*). The connection is weighted, so that a spike in the presynaptic neuron sends a current of 5 nanoamps (nA) to the excitatory synapse of the postsynaptic neuron, with a delay of 1 millisecond. The spikes and the membrane voltage are recorded, and the simulation is then run for 10 milliseconds. Graphs are then created of the membrane voltage and the spikes produced.

PyNN provides a number of standard neuron models. One of the most basic of these is known as the *Leaky Integrate and Fire* (LIF) model, and this is used above (*IF_curr_exp*). This models the neuron as a resistor and capacitor in parallel; as charge is received, this builds up in the capacitor, but then leaks out through the resistor. In addition, a *threshold* voltage is defined; if the voltage reaches this value, a spike is produced. For a time after this, known as the *refractory period*, the neuron is not allowed to spike again.

Once this period has passed, the neuron resumes operation as before. Additionally, the synapses are modelled using an exponential decay of the received current input (5 nA in the above example); the weight of the current is added over a number of timesteps, with the current decaying exponentially between each. A longer decay rate will result in more charge being added overall per spike that crosses the synapse.

In the above example, the default parameters of the *IF_curr_exp* are used. These are:

```
'cm': 1.0,         # The capacitance of the LIF neuron in nano-Farads
'tau_m': 20.0,     # The time-constant of the RC circuit, in milliseconds
'tau_refrac': 2.0, # The refractory period, in milliseconds
'v_reset': -70.0,  # The voltage to set the neuron at immediately after a spike
'v_rest': -65.0,   # The ambient rest voltage of the neuron
'v_thresh': -50.0, # The threshold voltage at which the neuron will spike
'tau_syn_E': 5.0,  # The excitatory input current decay time-constant
'tau_syn_I': 5.0,  # The inhibitory input current decay time-constant
'i_offset': 0.0,   # A base input current to add each timestep
```

PyNN supports both current-based models and conductance-based models. In conductance models, the input is measured in microSiemens, and the effect on the membrane voltage also varies with the current value of the membrane voltage; the higher the membrane voltage, the more input is required to cause a spike. This is modelled as the *reversal potential* of the synapse; when the membrane potential equals the reversal potential, no current will flow across the synapse. A conductance-based version of the LIF model is provided, which, in addition to the above parameters, also supports the following:

```
'e_rev_E': 0.,    # The reversal potential of the exponential synapse
'e_rev_I': -80.0  # The reversal potential of the inhibitory synapse
```

The initial value of the state variables of the neural model can also be set (such as the membrane voltage). This is done via the *initialize* function of the population, which takes the name of the state variable as a string (e.g. “v” for the membrane voltage), and the value to be assigned e.g. to set the voltage to -65.0mV:

```
pop.initialize(“v”, -65.0)
```

In PyNN, the neurons are declared in terms of a *population* of a number of neurons with similar properties. The *projection* between populations therefore has a *connector*, which describes the connectivity between the individual neurons in the populations. Some common connectors include:

- OneToOneConnector - each presynaptic neuron connects to one postsynaptic neuron (there should be the same number of neurons in each population) with weight *weights* and delay *delays*.
- AllToAllConnector - all presynaptic neurons connect to all postsynaptic neurons with weight *weights* and delay *delays*.
- FixedProbabilityConnector - each presynaptic neuron connects to each postsynaptic neuron with a given fixed probability *p_connect*, with weight *weights* and delay *delays*.
- FromListConnector - the exact connectivity is described by *conn_list*, which is a list of (*pre_synaptic_neuron_id, post_synaptic_neuron_id, weight, delay*)

Commonly, random weights and/or delays are used. To specify this, the value of the *weights* or *delays* of the connector are set to a *RandomDistribution* (note that the FromListConnector requires the specification of explicit weights and delays, and so does not support this; instead the *next()* method of the random distribution can be called to give random values for this connector). This supports several parameters via the *parameters* argument, depending on the value of the *distribution* argument which identifies the distribution type. The supported distributions include a ‘uniform’ distribution, with parameters of [minimum value, maximum value]; and a ‘normal’ distribution with parameters of [mean, standard deviation]. A *boundary* can also be specified as [*minimum, maximum*] to constrain the values generated (where an unbounded end can make use of -numpy.inf or numpy.inf); this is often useful for keeping the delays within

range allowed by the simulator. The *RandomDistribution* can also be used when specifying neural parameters, or when initialising state variables.

In addition to neuron models, the PyNN language also supports some utility models, which can be used to simulate inputs into the network with defined characteristics. These include:

- SpikeSourceArray - this sends spikes at predetermined intervals defined by *spike_times*. In general, PyNN forces each of the neurons in the population to spike at the same time, and so *spike_times* is an array of times, but sPyNNaker also allows *spike_times* to be an array of arrays, each defining the the times at which each neuron should spike e.g. *spike_times*=[[0], [1]] means that the first neuron will spike at 0ms and the second at 1ms.
- SpikeSourcePoisson - this sends spikes at random times with a mean rate of *rate* spikes per second, starting at time *start* (0.0ms by default) for a duration of *duration* milliseconds (the whole simulation by default).

Using PyNN with SpiNNaker

In addition to the above steps, sPyNNaker requires the additional step of configuration via the .spynnaker.cfg file to indicate which physical SpiNNaker machine is to be used. This file is located in your home directory, and the following properties must be configured:

```
[Machine]
machineName     = None
version         = None
```

The *machineName* refers to the host or IP address of your SpiNNaker board. For a 4-chip board that you have directly connected to your machine, this is *usually* (but not always) set to *192.168.240.253*, and the *version* is set to *3*, indicating a “SpiNN-3” board (often written on the board itself). Most 48-chip boards are given the IP address of *192.168.240.1* with a *version* of *5*.

The range of delays allowed when using sPyNNaker depends upon the timestep of the simulation. The range is 1 to 144 timesteps, so at 1ms timesteps, the range is 1.0ms to 144.0ms, and at 0.1ms, the range is 0.1ms to 14.4ms.

The default number of neurons that can be simulated on each core is 256; larger populations are split up into 256-neuron chunks automatically by the software. Note though that the cores are also used for other things, such as input sources, and delay extensions (which are used when any delay is more than 16 timesteps), reducing the number of cores available for neurons.

Spike-Time-Dependent Plasticity

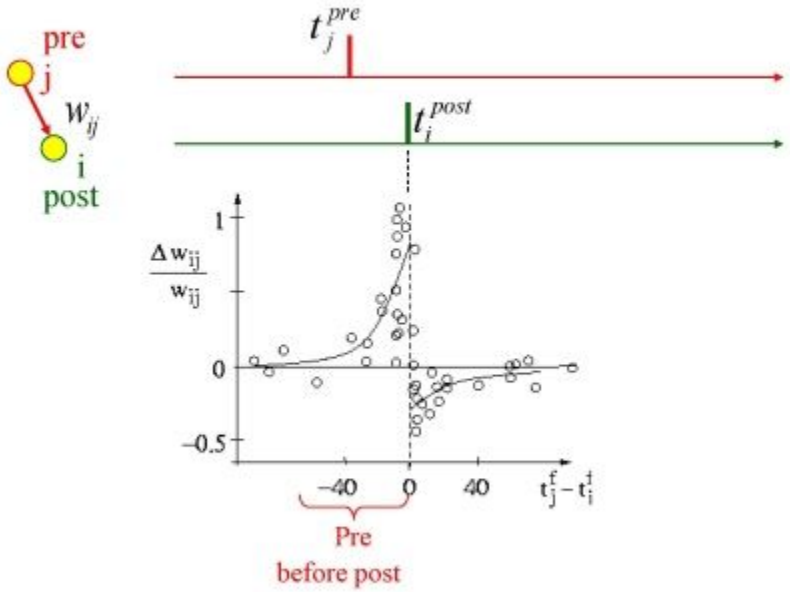
STDP plasticity is a form of learning that depends upon the timing between the spikes of two neurons connected by a synapse. It is believed to be the basis of learning and information storage in the human brain.

In the case where a presynaptic spike is followed closely by a postsynaptic spike, then it is presumed that the presynaptic neuron caused the spike in the postsynaptic neuron, and so the weight of the synapse between the neurons is increased. This is known as potentiation.

If a postsynaptic spike is emitted shortly before a presynaptic spike is emitted, then the presynaptic spike cannot have caused the postsynaptic spike, and so the weight of the synapse between the neurons is reduced. This is known as depression.

The size of the weight change depends on the relative timing of the presynaptic and postsynaptic spikes; in general, the change in weight drops off exponentially as the time between the spikes gets larger, as shown in the following figure [Sjöström and Gerstner (2010), Scholarpedia]. However, different experiments have

highlighted different behaviours depending on the conditions (e.g. [Graupner and Brunel (2012), PNAS]). Other authors have also suggested a correlation between triplets and quadruplets of presynaptic and postsynaptic spikes to trigger synaptic potentiation or depression.



STDP in PyNN

The steps for creating a network using STDP are much the same as previously described, with the main difference being that some of the projections are given an *synapse_dynamics* argument to describe the plasticity. Here is an example of the creation of a projection with STDP:

```
timing_rule = p.SpikePairRule(tau_plus=20.0, tau_minus=20.0)
weight_rule = p.AdditiveWeightDependence(
    w_max=5.0, w_min=0.0, A_plus=0.5, A_minus=0.5)

stdp_model = p.STDPMechanism(
    timing_dependence=timing_rule, weight_dependence=weight_rule)

stdp_projection = p.Projection(
    pre_pop, post_pop, p.OneToOneConnector(weights=0.0, delays=5.0),
    synapse_dynamics=p.SynapseDynamics(slow=stdp_model))
```

In this example, firstly the timing rule is created. In this case, it is a *SpikePairRule*, which means that the relative timing of the spikes that will be used to update the weights will be based on pairs of presynaptic and postsynaptic spikes. This rule has two required parameters, *tau_plus* and *tau_minus* which describe the respective exponential decay of the size of the weight update with the time between presynaptic and postsynaptic spikes. Note that the decay can be different for potentiation (defined by *tau_plus*) and depression (defined by *tau_minus*). This rule also accepts a parameter of *nearest* (by default this is *False*). If set to true, only the nearest pairs of spikes are considered when looking at weight updates.

The next thing defined is the weight update rule. In this case it is a *AdditiveWeightDependence*, which means that the weight will be updated by simply adding to the current weight. This rule requires the parameters *w_max* and *w_min*, which define the maximum and minimum weight of the synapse respectively, and the parameters *A_plus* and *A_minus* which define the maximum weight to respectively add during potentiation or subtract during depression. Note that the actual amount added or subtracted will depend additionally on the timing of the spikes, as determined by the timing rule.

In addition, there is also a *MultiplicativeWeightDependence* supported, which means that the weight change depends on the difference between the current weight and w_max for potentiation, and w_min for depression. The value of *A_plus* and *A_minus* are then respectively multiplied by this difference to give the

maximum weight change; again the actual value depends on the timing rule and the time between the spikes.

The timing and weight rules are combined into a single *STDPMechanism* object which describes the overall desired mechanism. This is finally specified as the *slow* argument of a *SynapseDynamics* object, and then added to the *Projection* using the *synapse_dynamics* argument. Note that the projection still requires the specification of a *Connector* which includes *weights* and *delays*. This connector is still used to describe the overall connectivity between the neurons of the pre- and post-populations, as well as the delay values which are unchanged by the STDP rules, and the initial weight values. It is preferable that the initial weights fall between *w_min* and *w_max*; it is not an error if they do not, but when the first update is performed, the weight will be changed to fall within this range.

Note: In the implementation of STDP on SpiNNaker, the plasticity mechanism is only activated when the second presynaptic spike is received at the postsynaptic neuron. Thus at least two presynaptic spikes are required for the mechanism to be activated.

Task 1.1: A simple neural network [Easy]

This task will create a very simple network from scratch, using some of the basic features of PyNN and SpiNNaker.

Write a network with a 1.0ms time step, consisting of two input source neurons connected to two current-based LIF neurons with default parameters, on a one-to-one basis, with a weight of 5.0 nA and a delay of 2ms. Have the first input neuron spike at time 0.0ms and the second spike at time 1.0ms. Run the simulation for 10 milliseconds. Record and plot the spikes received against time.

Task 1.2: Changing parameters [Easy]

This task will look at the parameters of the neurons and how changing the parameters will result in different network behaviour.

Using your previous script, set tau_syn_E to 1.0 in the IF_curr_exp neurons. Record the membrane voltage in addition to the spikes. Print the membrane voltage out after the simulation (you can plot it if you prefer, but you should note that the array returned from get_v() contains a list of [neuron_id, time, voltage] and so you will need to separate out the voltages of the individual neurons).

1. Did any of the neurons spike?
2. What was the peak membrane voltage of any of the neurons, compared to the default threshold voltage of -50mV?

Try increasing the weight of the connection and see what effect this has on the spikes and membrane voltage.

Task 2.1: Synfire Chain [Moderate]

This task will create a network known as a Synfire chain, where a neuron or set of neurons spike and cause activity in an ongoing chain of neurons or populations, which then repeats.

1. Setup the simulation to use 1ms timesteps.
2. Create an input population of 1 source spiking at 0.0ms.
3. Create a synfire population with 100 neurons.
4. With a FromListConnector, connect the input population to the first neuron of the synfire population, with a weight of 5nA and a delay of 1ms.
5. Using another FromListConnector, connect each neuron in the synfire population to the next neuron, with a weight of 5nA and a delay of 5ms.

6. Connect the last neuron in the synfire population to the first.
7. Record the spikes produced from the synfire populations.
8. Run the simulation for 2 seconds, and then retrieve and plot the spikes from the synfire population.

Task 2.2: Random Values [Easy]

Update the network above so that the delays in the connection between the synfire population and itself are generated from a uniform random distribution with values between 1.0 and 15.0. Update the run time to be 5 seconds.

Task 3.1: Balanced Random Cortex-like Network [Hard]

This task will create a network that this similar to part of the Cortex in the brain. This will take some input from outside of the network, representing other surrounding neurons in the form of poisson spike sources. These will then feed into an excitatory and an inhibitory network set up in a balanced random network. This will use distributions of weights and delays as would occur in the brain.

1. Set up the simulation to use 0.1ms timesteps.
2. Choose the number of neurons to be simulated in the network.
3. Create an excitatory population with 80% of the neurons and an inhibitory population with 20% of the neurons.
4. Create excitatory poisson stimulation population with 80% of the neurons and an inhibitory poisson stimulation population with 20% of the neurons, both with a rate of 1000Hz.
5. Create a one-to-one excitatory connection from the excitatory poisson stimulation population to the excitatory population with a weight of 0.1nA and a delay of 1.0ms.
6. Create a similar excitatory connection from the inhibitory poisson stimulation population to the inhibitory population.
7. Create an excitatory connection from the excitatory population to the inhibitory population with a fixed probability of connection of 0.1, and using a normal distribution of weights with a mean of 0.1 and standard deviation of 0.1 (remember to add a boundary to make the weights positive) and a normal distribution of delays with a mean of 1.5 and standard deviation of 0.75 (remember to add a boundary to keep the delays within the allowed range on SpiNNaker).
8. Create a similar connection between the excitatory population and itself.
9. Create an inhibitory connection from the inhibitory population to the excitatory population with a fixed probability of connection of 0.1, and using a normal distribution of weights with a mean of -0.4 and standard deviation of 0.1 (remember to add a boundary to make the weights negative) and a normal distribution of delays with a mean of 0.75 and standard deviation of 0.375 (remember to add a boundary to keep the delays within the allowed range on SpiNNaker).
10. Create a similar connection between the inhibitory population and itself.
11. Initialize the membrane voltages of the excitatory and inhibitory populations to a uniform random number between -65.0 and -55.0.
12. Record the spikes from the excitatory population.
13. Run the simulation for 1 or more seconds.
14. Retrieve and plot the spikes.

The graph should show what is known as Asynchronous Irregular spiking activity - this means that the neurons in the population don't spike very often and when they do, it is not at the same time as other neurons in the population.

Task 3.2: Network Behavior [Moderate]

Note in the above network that the weight of the inputs is the same as the mean weight of the excitatory connections (0.1nA) and that the mean weight of the inhibitory connections is 4 times this value (-0.4nA). Try setting the excitatory connection mean weight and input weights to 0.11nA and the inhibitory mean

weight to -0.44nA, and see how this affects the behavior. What other behavior can you get out of the network by adjusting the weights?

Task 4.1: STDP Network [Easy]

This task will create a simple network involving STDP learning rules.

Write a network with a 1.0ms time step consisting of two single-neuron populations connected with an STDP synapse using a spike pair rule and additive weight dependency, and initial weights of 0. Stimulate each of the neurons with a spike source array with times of your choice, with the times for stimulating the first neuron being slightly before the times stimulating the second neuron (e.g. 2ms or more), ensuring the times are far enough apart not to cause depression (compare the spacing in time with the tau_plus and tau_minus settings); note that a weight of 5.0 should be enough to force an IF_curr_exp neuron to fire with the default parameters. Add a few extra times at the end of the run for stimulating the first neuron. Run the network for a number of milliseconds and extract the spike times of the neurons and the weights.

You should be able to see that the weights have changed from the starting values, and that by the end of the simulation, the second neuron should spike shortly after the first neuron.

Task 4.2: STDP Parameters [Easy]

Alter the parameters of the STDP connection, and the relative timing of the spikes. Try starting with a large initial weight and see if you can get the weight to reduce using the relative timing of the spikes.

Task 5: STDP Curve [Hard]

This task will attempt to plot an STDP curve, showing how the weight change varies with timing between spikes.

1. Set up the simulation to use a 1ms time step.
2. Create a population of 100 presynaptic neurons.
3. Create a spike source array population of 100 sources connected to the presynaptic population. Set the spikes in the arrays so that each spikes twice 200ms apart, and that the first spike for each is 1ms after the first spike of the last e.g. [[0, 200], [1, 201], …] (hint: you can do this with a list comprehension).
4. Create a population of 100 postsynaptic neurons.
5. Create a spike source array connected to the postsynaptic neurons all spiking at 50ms.
6. Connect the presynaptic population to the postsynaptic population with an STDP projection with an initial weight of 0.5 and a maximum of 1 and minimum of 0.
7. Record the presynaptic and postsynaptic populations.
8. Run the simulation for long enough for all spikes to occur, and get the weights from the STDP projection.
9. Draw a graph of the weight changes from the initial weight value against the difference in presynaptic and postsynaptic neurons (hint: the presynaptic neurons should spike twice but the postsynaptic should only spike once; you are looking for the first spike from each presynaptic neuron).

Graph Front End Lab Manual

The task is to build a python program that uses the Graph Front End (GFE).

Summary

This python script should use the GFE to instantiate a working example of the Conway's Game of Life[1]. Conway's Game of Life consist of a 2D fabric of cells, each of which has 2 states. These states are either Alive or Dead, and switching between these two states is decided upon the states of their 8 neighbouring cells.

The rules which dictate the changing of state are as follows:

1. Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if caused by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The application needs to be able to handle different initial states of the cells within the 2d fabric, but the default states for tasks 1 to 8 should look like Figure 1 and for task 9 to 13 should look like Figure 2.

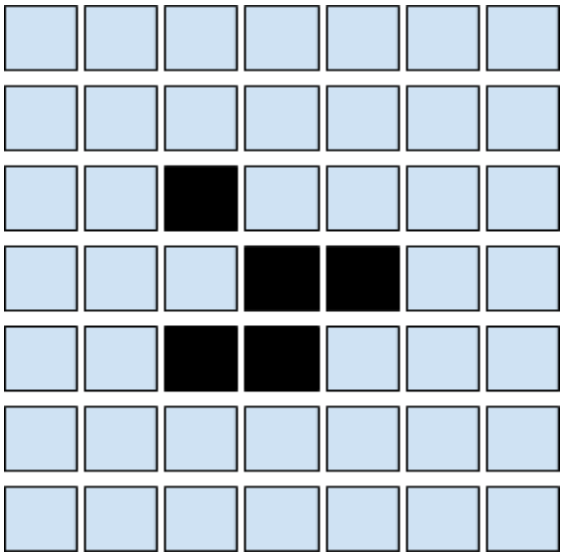


Figure 1: Basic initial state (7 by 7 grid)

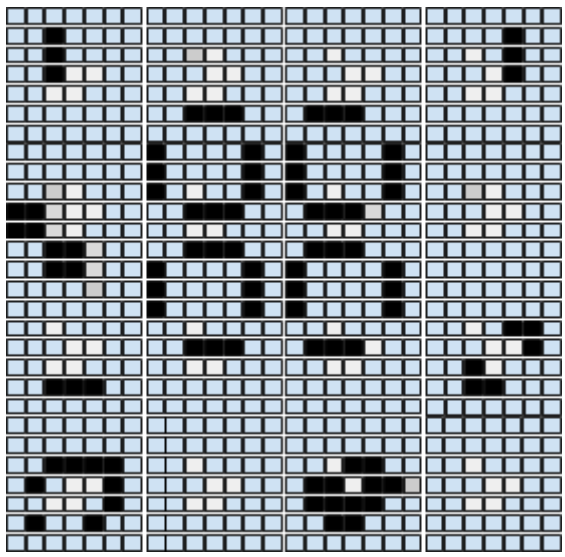


Figure 2: Advanced State (28 by 28 grid)

[1] https://en.wikipedia.org/wiki/Conway's_Game_of_Life

Step 1 (Easy)

Create a python class which will represent a Conway Cell.

Step 2 (Easy)

Build a python script which builds a collection of Conway cells (as vertices) in the GFE machine graph to form a 7x7 grid (hint: use the add_machine_vertex_instance() functionality supplied with the GFE __init__.py interface to add vertex instances). Add edges between the cells.

Step 3 (Medium)

Build the c code that represents the functionality that will run on the SpiNNaker machine. It'll be easier to use the interfaces provided in simulation.h and data_specification.h.

At this point running the script should not produce errors, but you won't be able to tell what's happening inside.

Step 4 (Medium)

Add a data region for storing the state per timer tick iteration in SDRAM into both the c and python class. Add code to store the data in C and retrieve it from the machine in python.

Step 5 (Easy)

Build a simple text-based visualiser to replay a simulation run using the stored state.

At this point you should be able to run the simulation and get a textual display of the state of the simulation per timer tick.

Step 6 (Hard)

Stream the state of the simulation to the host PC during the simulation run. Display the output as text as the simulation runs.

Step 7 (Medium)

Use the **tubogrid** perl application supplied within spinnaker_tools to create a live visualisation of the simulation, or build your own.

Step 8 (Easy)

Try building a 28x28 grid of cells and see what happens. Can you explain why it doesn't work? What ways could you go about making it work (hint: there's at least 3 ways of doing this)?

Step 9 (Fiendishly Hard)

Convert your application so that instead of using machine vertices, you use an application vertex to represent the entire grid of cells. Note that this will require you to receive and interpret several ids from each base routing key (hint: sPyNNaker does this using something called a population table to map between a base key and a block of connections).

Step 10 (Easy)

Run your new simulation with both sets of initial parameters. Try scaling up so that you hit the limits of the system to simulate the application, and therefore make the CPU calculation correct so that the partitioner will stop you going over the limits.

Step 11 (Medium)

Upgrade **turbogrid** or your own visualisation to use the database so that it auto configures itself for the shape of the application space.

Step 12 (Epically Hard)

Build your own application for SpiNNaker using the GFE front end.

Simple Data Input Output and Visualisation on Spinnaker - Lab Manual

1. Introduction

This manual will introduce you to the basics of live retrieval and injection of data (in the form of spikes) for PyNN scripts that are running on SpiNNaker neuromorphic hardware.

2. Installation

In addition to sPyNNaker, the sPyNNakerExternalDevicesPlugin must also be installed.

3. PyNN Support

This section discusses the standard support from PyNN related to spike injection and retrieval.

3.1 Output

The standard support for data output for a platform such as SpiNNaker, through the PyNN language, is to use the methods record(), record_v(), for declaring the need to record, and get_Spikes(), get_v(), for retrieval of the specific data.

The issue with the get functions are that they are called after run() completes, and therefore are not live, and so not able to interact with an external device running in real-time. In the current implementation of sPyNNaker, all of the data declared to be recorded via record(), record_v(), is stored on the SDRAM of the chips that the corresponding populations were placed on. By writing the data to SDRAM, the data is stored locally and therefore is guaranteed to be read at some point in the future. In the current implementation, if the memory requirements for recording cannot be met, the model will be run for less time, paused whilst the data is extracted, and then resumed. This may be repeated a number of times until the whole simulation has completed.

When used with an external simulation, it is possible to call run a number times, extracting the data between each run and passing it to an external simulation. This mode of operation will not work if the external device or simulation cannot also be paused.

3.2 Input

The standard support for data input for a platform such as SpiNNaker, through the PyNN language, is to use the neural models SpikeSourceArray and SpikeSourcePoisson. The issue with both of these models is that they are either random rate based (the spikeSourcePoisson) or have to be supplied in advance with all the spikes to be sent (SpikeSourceArray). As with the output of spikes, it is possible to change the input spikes of a SpikeSourceArray between successive calls to run(). Again, this will only work if the external device or simulation can be paused.

4. External Device Plugin Support

As stated previously, the issue with this is that PyNN 0.7 expects its **run()** method to block for the entire time of the run, and therefore it is impossible to set up a real time extraction or retrieval of data via this FrontEnd (sPyNNaker), and has no current support for live retrieval or live injection.

It is worth noting that future releases of PyNN may use the MUSIC interface to support live injection and retrieval of spikes, but the current software version of sPyNNaker only supports PyNN 0.7 and therefore there is no built in support.

To compensate for this, the sPyNNakerExternalDevicesPlugin module was created that contains support for live injection and retrieval of spikes from a running PyNN 0.7 simulation during the simulation, whilst still maintaining the real-time operation of the simulation.

4.1 Live Output

To activate live retrieval from a given population, the command

activate_live_output_for(<Population_object>**)**

is used. This informs the sPyNNaker backend to add the supporting utility model (Live packet gatherer) into the graph object (which sPyNNaker uses to represent your PyNN neural models) and an edge between your population and the associate LPG for your ports.

Other parameters for the **activate_live_output_for()** function are defined below:

Parameter	**Description**
port	The port number to receive packets from the SpiNNaker machine.
database_notify_host	The hostname for the database notification protocol; by default the localhost is used, but any external host could act as a receiver, provided it can read the file system that the database is written to.
database_notify_port_num	The port number for the database notification protocol; by default database notifications are sent to port 19998. However this can be changed if there is more than one listener, or a separate listener for each population.
database_ack_port_num	The port number that the database notification protocol will listen to, to receive the acknowledgement packet that the database has been read. By default this is 19999. It is unlikely that this needs to be changed.

4.2 Live Injection

To activate the live injection functionality, you need to instantiate a new neural model (called a SpikeInjector) which is located in **spynnaker_external_devices_plugin.pyNN.SpikeInjector**

The **SpikeInjector** is considered as any other neural model in PyNN, so you can build a population with a number of neurons etc in the normal way, as shown below:

```
injector_forward = Frontend.Population(
       5, ExternalDevices.SpikeInjector, [‘port’: 12367],
       label='spike_injector_forward')
```

The key parameters of the **SpikeInjector** are as follows:

Parameter	**Description**
port	The port that packets are going to be sent to on the SpiNNaker system - must be one per injector, but any port other than 17893 or 54321 can be used (these are reserved for SpiNNaker operations).
virtual_key	The base routing key that the spike injector is going to use for routing. **This parameter is optional.**

4.3 Python Live reciever

The following block of code creates a live packet receiver to receive spikes from a live simulation:

```
1   # declare python code when received spikes for a timer tick
2   def receive_spikes(label, time, neuron_ids):
3       for neuron_id in neuron_ids:
4       print “Received spike at time {} from {}-{}”.format(
5           time, label, neuron_id)
6
7   # import python live spike connection
8   from spynnaker_external_devices_plugin.pyNN.connections.\
9       spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
10
11  # set up python live spike connection
12  live_spikes_connection = SpynnakerLiveSpikesConnection(
13      receive_labels=[“receiver”])
14
15  # register python receiver with live spike connection
16  live_spikes_connection.add_receive_callback(“receiver”, receive_spikes)
```

1. Lines 1 to 5 creates a function that takes as its input all the neuron ids that fired at a specific time, from the population with the given label. From here, it generates a print message for each neuron.
2. Lines 7 to 9 imports the python support for live injection/live retrieval. The SpynnakerLiveSpikesConnection handles both live retrieval and live injection.
3. Lines 11 to 13 instantiates the SpynnakerLiveSpikesConnection, and informs the connection that it will receive data under the label “receiver”.
4. Lines 15 to 16 informs the connection that any packets being received with the “receiver” label need to be forwarded to the function receive_spikes defined on lines 1 to 5.

This script must be run in advance of the script that sets up the simulation. The SpynnakerLiveSpikesConnection will listen for the simulation script to complete the setup operations and so starts synchronized with the simulation. It is possible to run the reception of spikes within the same script as the simulation; to do this, ensure that the above code is placed before the call to run().

If you need more than one SpynnakerLiveSpikesConnection on the same host, the connection can take an additional parameter specifying the local port to listen on for notifications from the simulation, by specifying the local_port parameter in the constructor e.g.:

```
live_spikes_connection_1 = SpynnakerLiveSpikesConnection(
    receive_labels=[“receiver”], local_port=19996)
live_spikes_connection_2 = SpynnakerLiveSpikesConnection(
    receive_labels=[“receiver_2”], local_port=19997)
```

Note that you must then also tell the simulation side that these ports are in use. This can be done when calling activate_live_output_for for the population by specifying the database_notify_port_num parameter e.g.

```
activate_live_output_for(receiver, database_notify_port_num=19996)
activate_live_output_for(receiver_2, database_notify_port_num=19997)
```

4.4 Python Live injector

The following block of code creates a live packet injector:

```
1  # create python injector
2  def send_spike(label, sender):
3      sender.send_spike(label, 0, send_full_keys=True)
5
6  # import python injector connection
7  from spynnaker_external_devices_plugin.pyNN.connections.\
8  spynnaker_live_spikes_connection import SpynnakerLiveSpikesConnection
9
10 # set up python injector connection
11 live_spikes_connection = SpynnakerLiveSpikesConnection(
12     send_labels=[“spike_sender”])
13
14 # register python injector with injector connection
```

```
15 live_spikes_connection.add_start_callback("spike_sender", send_spike)
```

1. Lines 1 to 3 create a function that will be called when the simulation starts, allowing the synchronized sending of spikes.
2. Lines 6 to 8 imports the python support for live injection/live retrieval. The SpynnakerLiveSpikesConnection handles both live retrieval and live injection.
3. Lines 10 to 12 instantiates the SpynnakerLiveSpikesConnection, and informs the connection it will inject data via the label spike_sender.
4. Lines 14 to 15 informs the connection that when the simulation starts, to call the send_spike function defined on lines 1 to 3.

As with the live reception script, this must be called before the simulation script, or before run() in the simulation script.

4.5 C++ Implementation of SpyNNakerLiveSpikesConnection and Visualiser

The host C++ version of the Python "SpynnakerLiveSpikesConnection" and example visualiser is currently available from the following locations:

https://spinnakermanchester.github.io/latest/visualiser_code_zip.html
https://spinnakermanchester.github.io/latest/visualiser_code_tar_gz.html

This source code must be compiled before use, and depends on the pthread and sqlite libraries for the library itself, and the freeglut and opengl libraries for the example visualiser application. A Makefile exists at the top level folder which will make both the spynnaker_external_device_lib library and the example visualiser, but each can be made separately by running make in the appropriate subdirectory.

Dependency Installation

On OSX, using Macports, you can install the dependencies as follows:

sudo port install freeglut sqlite3

On Linux, you can install the dependencies as follows (depending on if you are using Fedora or Ubuntu):

sudo yum install
sudo apt-get install

On Windows, the dependencies are included.

spynnaker_external_device_lib

The C++ implementation is designed to be similar to the Python implementation. A number of sample applications are provided within the spynnaker_external_device_lib/examples folder which show how the API can be used.

c_based_visualiser_framework

This contains an example visualiser for producing a spike raster plot, and is based on the spynnaker_external_device_lib.

The visualiser application can accept 4 parameters. These are defined below:

Parameter	Description
-colour_map	Path to a file containing the population labels to receive, and their associated colours. This must be specified.
-hand_shake_port	Optional port which the visualiser will listen to for database handshake (default is 19998).
-database	Optional file path to where the database is located, if needed for manual configuration.
-remote_host	Optional remote host address of the SpiNNaker board, which allows port triggering if allowed by your firewall.

7.1 colour_map file format

The colour_map file consists of a collection of lines, where each line contains 4 values separated by tabs. These values, in order are:

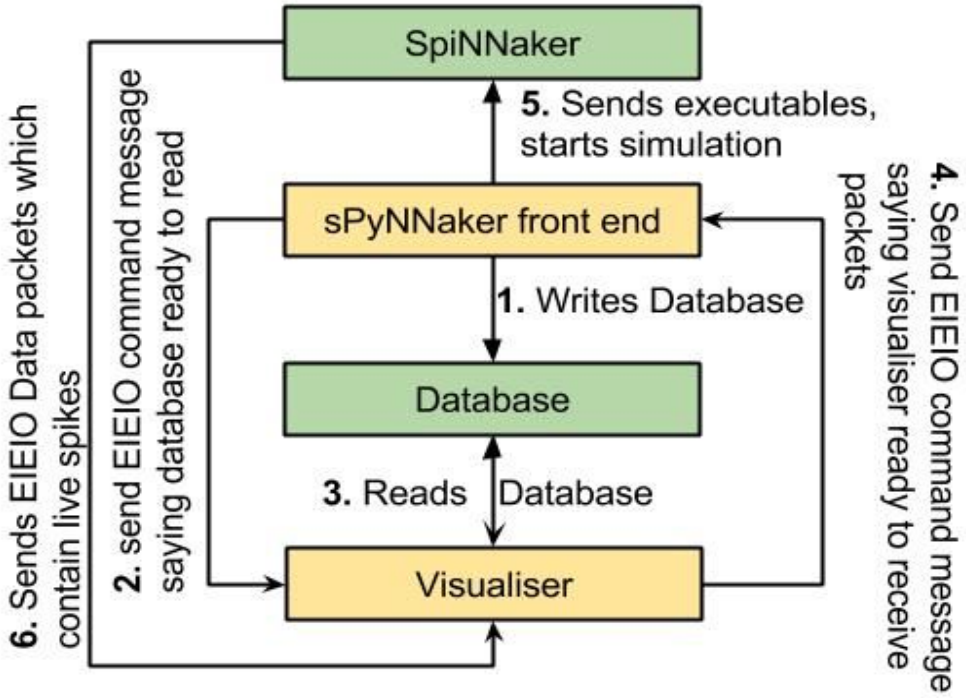
1. The population label.
2. The red colour value.
3. The green colour value.
4. The blue colour value.

An example file is shown below:

```
spike_forward 0	0	255
spike_backwards	0	255	0
```

5. Database Notification protocol

The support built behind all this software is a simple notification protocol on a database that's written during compilation time. The notification protocol is illustrated below:



The steps within the notification protocol are defined below:

1. The sPyNNaker front end writes a database that contains all the data objects generated from sPyNNaker during the compilation process.
2. The notification protocol sends a message to all the notification protocol listeners containing the path to the database to be read. The SpynnakerLiveSpikesConnection Python and C implementations are set up to receive this message.
3. These devices then read the database to determine the information required. This includes the port to listen on to receive live output spikes, the port to send like input spikes to, and the mapping between SpiNNaker routing keys and neuron ids.
4. Once these devices have read the database, they notify the sPyNNaker front end that they are ready for the simulation to start.
5. Once all devices have notified the sPyNNaker front end, the simulation begins. The sPyNNaker front end also notifies the devices when the simulation has actually started, in case it was still loading data when they became ready.
6. The SpiNNaker machine transmits live spike output packets and receives live spike input packets.

6. Caveats

To use the live injection and retrieval functionality only supports the use of the Ethernet connection, which means that there is a limited bandwidth of a maximum of approx 30 MB/s. This bandwidth is shared between both types of functionality, as well as system support for certain types of neural models, such as the SpikeSourceArray.

Furthermore, this functionality depends upon the lossy communication fabric of the SpiNNaker machine. This means that even though a neuron fires a spike you may not see it via the live retrieval functionality. If you need to ensure you receive every packet that has been transmitted, we recommend using the standard PyNN functionality.

By using this functionality, you are making your script non portable between different simulators. The activate_live_output_for(<pop_object>) and SpikeInjector models are not supported by other PyNN backends (such as Nest, Brian etc).

Finally, this functionality uses a number of additional SpiNNaker cores. Therefore a network which would just fit onto your SpiNNaker machine before would likely fail to fit on the machine when these functionalities are added in.

8. Tasks

Task 1.1: A synfire chain with injected spike via python injector [Easy]

This task will create a synfire chain which is stimulated from a injector spike generated on host and then injected into the simulation. Start with the synfire chain from PyNNExamples.

1. Remove the spike source array population.
2. Replace it with the SpikeInjector population.
3. Build a python injector function.
4. Import and instantiate an SpynnakerLiveSpikesConnection connection.
5. link a start callback to the python injector function.

Task 1.2: A synfire chain with live streaming via the python receiver [Easy]

Start with the synfire chain from PyNNExamples.

1. Call activate_live_output_for(<pop_object>) on the synfire population.
2. Build a python receiver function that prints out the neuron ids for the population.
3. Import and instantiate a SpynnakerLiveSpikesConnection connection.
4. Link a receive callback to the python receiver function and print when a spike is received.

Task 1.3: A synfire chain with live injection and streaming via python [Easy]

Take the code from the previous 2 tasks and integrate them together to produce one that injects and streams the packets back to the terminal.

1. Remember that you can use both the recieve_labels and send_labels of the same SpynnakerLiveSpikesConnection.

Task 1.4: A synfire chain with live injection via python and live streaming via the c visualiser [Medium]

Take the code from the previous task and remove the python receiver code (or don't if you feel confident) and activate the visualizer to take the packets the original python receiver code processed.

1. Remember to compile the visualiser
2. Remember to generate the correct colour_map
3. Remember to remove the python receiver code (or don't if you're feeling confident).

Task 1.5: 2 Synfire chains which set each other off using python injectors whilst still using the c visualiser [Very Hard]

Take the code from the previous task and modify it so that there are two synfire populations which are tied to one injector population. Modify the receive function so that it contains some logic that fires the second neuron when the last neuron in the first synfire fires, and does the same when the last neuron for the second synfire sets off some neuron id of the first synfire chain.

1. you will need to change the number of neurons the spike injector contains.
2. You will need to change the connector from the spike injector and each synfire population.
3. You will need to modify the receive function, and add a global variable for the SpynnakerLiveSpikesConnection.
4. You'll need at least 2 SpynnakerLiveSpikesConnection and multiple activate_live_output_for(<pop_objevt>) for each population.
5. Remember that each population can only be tied to one LivePacketGatherer, so to visualise and do closed loop systems require more populations.
6. You will need to modify the c visualiser colour_map to take into account the new synfire population.

Task 1.6: 2 Synfire chains which set each other off using python injectors and live retrieval with 2 visualiser instances [Very Hard/Easy]

This task takes everything you've learnt so far and raises the level. Using the code from the previous task. Create two visualiser instances, each of which only processes one synfire population.

1. Remember all the lessons from the previous tasks.
2. Remember to change the ports on the activate_live_output_for(<pop_object>) accordingly.
3. You will need to create at least 2 SpynnakerLiveSpikesConnection's. But it might be worth starting with 3 and reducing it to two once you've got it working.
4. Remember the different colour_maps

Task 2.1: A simple synfire chain with a injected spike via c injector [Easy]

This task requires that you replace the injector from task 1.1 with a c injector.

1. Remember to import the correct header file.
2. Remember to use c syntax.

Task 2.2: A simple synfire chain with live streaming via the c receiver [Easy]

This task requires that you replace the receiver from task 1.2 with a c receiver.

1. Remember to import the correct header file.

2. Remember to use c syntax.

Task 2.3: A simple synfire chain with live injection and live streaming via C [Easy]

This task requires that you replace the injector and receiver from task 1.3 with a c injector and receiver.

1. Remember to import the correct header file.
2. Remember to use c syntax.

Task 2.4: A simple synfire chain with live injection via c and live streaming via the c visualiser [Medium]

This task requires that you replace the injector from task 1.4 with a c injector and to set up the visualiser.

1. Remember to import the correct header file.
2. Remember to use c syntax.
3. Remember to set up the visualiser correctly.

Task 2.5: 2 Synfire chains which set each other off using c injectors [Medium]

This task requires that you replace the injectors and receivers from task 1.5 with c injectors and receivers and to set up the visualiser.

1. Remember to import the correct header file.
2. Remember to use c syntax.

Task 2.6: 2 Synfire chains which set each other off using c injectors and live retrieval with 2 visualiser instances [Hard]

This task requires that you replace the injectors and receivers from task 1.6 with c injectors and receivers and to set up the visualiser.

1. Remember to import the correct header file.
2. Remember to use c syntax.
3. Remember to set up the visualisers correctly.

Task 3: Create some model which uses all interfaces [Very Hard]

This task is the merging of all the functionalities covered in this lab manual. Take the codes from both task 2.6 and 1.6 and integrate them together so that:

1. One injector is controlled by the c code, whilst another is done via the python interface.
2. Still uses 2 visualisers to stream the results.
3. Uses the python receive interface to count 5 firings of a given neuron id and then changes the neuron stimulated by the python injector.

Hint: remember to keep a global connection object for the python codes.

Creating New Neuron Models for SpiNNaker

Introduction

This manual will guide you in the creation of new neuron models to be run on SpiNNaker. This includes the C code that will be compiled to run on the SpiNNaker hardware, as well as the Python code which interacts with the PyNN script to configure the model.

Installation

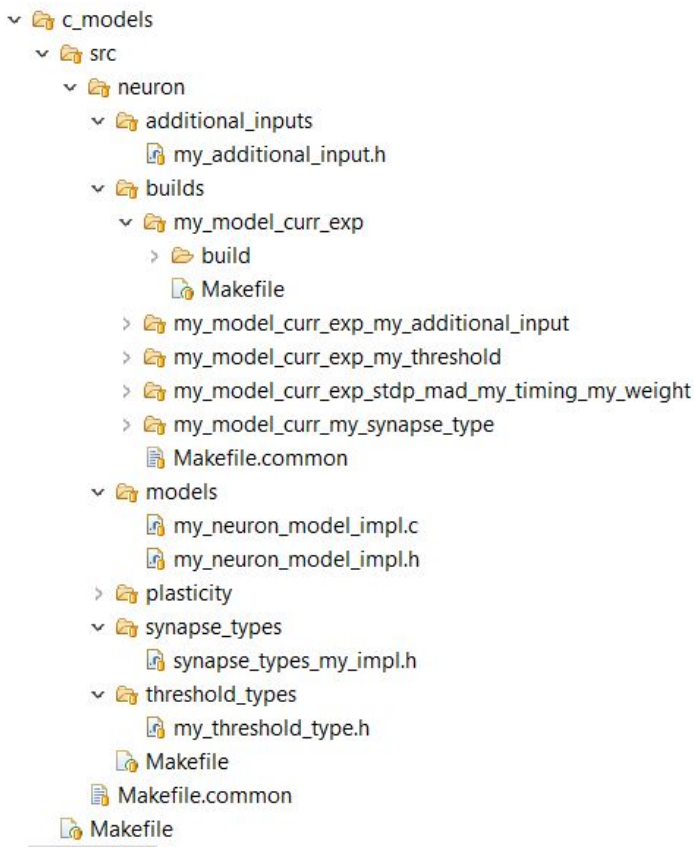
In order to create new models, you will need to ensure that you have set up a development environment suitable for compiling C code for SpiNNaker. This can be done by following the instructions here:

http://spinnakermanchester.github.io/latest/spynnaker_extensions.html

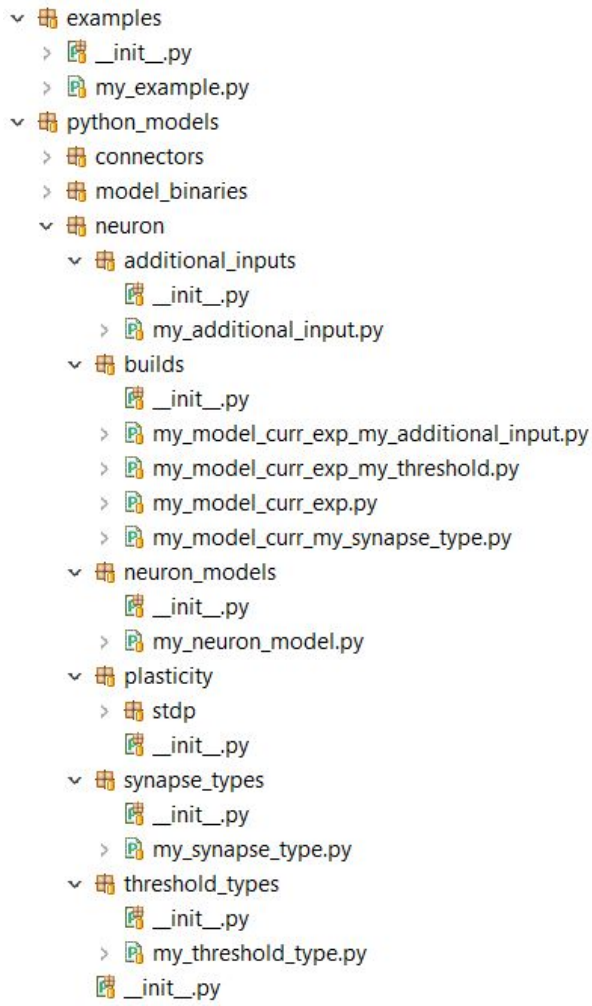
Project Layout

The recommended layout for a new model project is shown below; this example shows a model called "my_model", with current-based exponential synapses. It is recommended that the C and Python code is kept in the same project to help keep them synchronized.

C code



Python code



This template structure can be downloaded from one of the following locations:
https://spinnakermanchester.github.io/latest/spynnaker_new_model_template_zip.html
https://spinnakermanchester.github.io/latest/spynnaker_new_model_template_tar_gz.html

C model builds

All neuron builds consist of a collection of components which when connected together produce a complete neural model. These components are defined in **Table 1**.

Component	Definition
Input component	The type of input the model takes. Currently there are conductance and current based inputs supported by sPyNNaker. It is possible to define other input types, but this is not described in this tutorial.
Synapse type component	The synapse type controls the shaping of the synapses in response to the input weights. Within sPyNNaker, support so far includes exponential synapses (with one excitatory and one inhibitory synapse per neuron which decay exponentially with a configured time-constant) and dual-excitatory exponential synapses (with 2 separate excitatory synapses and one inhibitory synapse per neuron, decaying as per the previous type).
Threshold component	Determines the threshold of the membrane voltage which determines when the neuron spikes. Currently the only implementation is a static threshold.
Additional input component	Any additional input current that might be based on the membrane voltage or other parameters. This is currently only implemented in the SpyNNakerExtraModelsPlugin.
Neuron model component	Determines how the neuron state changes over time and the outputs the current membrane voltage of the neuron. Currently there are IZK and LIF implementations supported by sPyNNaker.
Synapse dynamics component	Determines how plasticity works within the model. sPyNNaker implements a model for no plasticity (i.e. static dynamics), and two different STDP dynamics models. Only static dynamics are considered in this tutorial.

Table 1: Different supported components

Each build is stored within its own folder in the c_models->src->neuron->builds directory. Within each build is a c *Makefile* which describes the separate components required to build that specific form of Neuron model.

If we look at the simple my_model_curr_exp's makefile located in:

c_models->src->neuron->builds->my_model_curr_exp->Makefile

we can see the lines show in **Code 1**.

```
1.  APP = $(notdir $(CURDIR))
2.  BUILD_DIR = build/
3.
4.  NEURON_MODEL = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.c
5.  NEURON_MODEL_H = $(EXTRA_SRC_DIR)/neuron/models/my_neuron_model_impl.h
6.  INPUT_TYPE_H = $(SOURCE_DIR)/neuron/input_types/input_type_current.h
7.  THRESHOLD_TYPE_H = $(SOURCE_DIR)/neuron/threshold_types/threshold_type_static.h
8.  SYNAPSE_TYPE_H = $(SOURCE_DIR)/neuron/synapse_types/synapse_types_exponential_impl.h
9.  SYNAPSE_DYNAMICS = $(SOURCE_DIR)/neuron/plasticity/synapse_dynamics_static_impl.c
10.
11. include ../Makefile.common
```

Code 1: my_model_curr_exp's Makefile

- Line 1 declares the name of the APP - here we are using the name of the current directory. The aplx extension is added automatically.
- Line 2 declares the directory in which the model will be built. This is where object files and other intermediate files are stored; the final aplx location is determined in Makefile.common (see later).
- Lines 4 and 5 states the files that make up the neuron model component (described in **Table 1**) used for this model build (both the .c and .h files are needed). Note that these are stated to be in the $(EXTRA_SRC_DIR) folder - this is declared to be the c_models/src folder within the archive within Makefile.common. The sPyNNaker standard source files are declared to be within $(SOURCE_DIR), and these are used by other components.
- Line 6 states the input type component (described in **Table 1**) for this model. Input types are implemented entirely in a header file.
- Line 7 states the threshold type component (described in **Table 1**) for this model. Threshold types are implemented entirely in a header file.
- Line 8 states the synapse type component (described in **Table 1**) for this model. Synapse types are implemented entirely in a header file.
- Line 9 states the synapse dynamics component (described in **Table 1)** for this model.
- Line 11 tells the make system to import the next level up Makefile so that it can detect where the rest of the code needed to be linked in can be found.

Other Makefile instances might also include TIMING_DEPENDENCE_H and WEIGHT_DEPENDENCE_H; these are used when the synapse dynamics includes plasticity. A tutorial on how to add new plasticity is covered here.

To make a new Neuron model build, you must either:

1. Create a copy of the example builds discussed above,
2. Modify the names and component listings,
3. Mody Line 1 of the Makefile located in src->neuron->Makefile so that it includes your new build.

Or:

1. Change the template's component listings directly.

Compiling a new model

Once the Makefile has been created, you can build the binary by simply changing to the directory containing the Makefile and typing:

```
make
```

As the build relies on header files that are not explicitly specified in the Makefile, some of the changes that you make may require you to clean the build before building it, by running

```
make clean
```

Finally, you can also build the application in debug mode by typing:

```
make SPYNNAKER_DEBUG=DEBUG
```

This will enable the log_debug statements in the code, which print out information to the iobuf buffers on the SpiNNaker machine. By default, the tools won't extract the printed error messages. To enable this behaviour, you can add the following to your .spynnaker.cfg file:

```
[Reports]
extract_iobuf=True
```

The “iobuf” messages will then be downloaded after the execution is complete. These are stored relative to your executing script in reports/latest/provenance_data/; each is a .txt file containing any output printed from your program.

C code file interfaces

The rest of this section goes through the different components interfaces and tries to explain what each one does, for the case where you need to create a new component for your neuron build.

Neuron Models

The C header file defines:

- The neuron data structure `neuron_t`. This includes the parameters and state for each neuron to be executed on a core. This commonly includes the membrane voltage of the neuron, as well as an offset input current.
- The global parameters data structure `global_neuron_params_t`. This includes parameters that are shared across all neurons within a population. This might include such things as the time step of the simulation.

See neuron_model_my_model_curr_exp.h in the template for an example of a header file. Comments show where the file should be updated to create your own model.

The C code file defines the functions that make up the interface of the neuron API. Note that pointer types are automatically created for the data structures defined in the header as follows:

```
                    neuron_t * → neuron_pointer_t
  global_neuron_params_t * → global_neuron_params_pointer_t
```

- ```
  void neuron_model_set_global_neuron_params(
        global_neuron_params_pointer_t params)
  ```
  This function is used to set the global parameters after they have been read by the initialization function. This would often be used to store the parameters in a static variable for later use.

- ```
  state_t neuron_model_state_update(
        input_t exc_input, input_t inh_input,
        input_t external_bias, neuron_pointer_t neuron)
  ```
  This function takes the excitatory and inhibitory input; any external bias input (used in some plasticity models); and a neuron data structure; and uses these to compute the new state of the given neuron at this timestep. This function is where any differential equation solving should be implemented. After the state update, the function should return the value of the membrane voltage. Note that the input will always be presented as current - conductance input is converted to current input in the input type. Additionally, the input values are all positive, including the inhibitory input; thus if the total input current is being considered, the inhibitory input current should be subtracted from the excitatory input current.

- `state_t neuron_model_get_membrane_voltage(neuron_pointer_t neuron)`
  This function should return the membrane voltage of the neuron from the given neuron structure. This may simply return the value of a variable in the structure, or it might perform a more complex calculation to obtain the membrane voltage. The value returned is used for the recording of the membrane voltage in the simulation, and is taken before the state update is performed.

- `void neuron_model_has_spiked(neuron_pointer_t neuron);`

This function is used to reset neuron parameters after it has spiked. It is called only if the membrane voltage value returned from `neuron_model_state_update` is determined to be above the threshold determined by the threshold type.

- `void neuron_model_print_parameters(restrict neuron_pointer_t neuron)`
  This function is only used when the neuron model is compiled in "debug" mode. It should use the "log_debug" function to print each of the parameters of the neuron that don't change during the run, and that might be useful in debugging.

- `void neuron_model_print_state_variables(restrict neuron_pointer_t neuron)`
  This function is only used when the neuron model is compiled in "debug" mode. It should use the "log_debug" function to print each of the state variables of the neuron that change during a run and that might be useful in debugging.

See neuron_model_my_impl.c in the template for an example of an implementation of the neuron interface.

A number of other modules are available for use for performing mathematical functions as part of the neuron state update. The `spinn_common` library provides a number of efficient fixed-point implementations of common functions. This includes `random.h`, which provides random number generation, `normal.h`, which provides normal distributions, `exp.h`, which provides an exp function and `log.h` which provides a log function.

Synapse types

The synapse type header file defines the `synapse_param_t` data structure that determines the parameters required for shaping the synaptic input. For example, this might be done to compensate for the valve behaviour of a synapse in biology (spike goes in, synapse opens, then closes slowly). The parameters for all the synaptic inputs for a single neuron need to be defined in this structure; for example, if there are different parameters for excitatory and inhibitory neurons, both of these parameters must be explicitly defined in this structure. The structure might also contain parameters for computing the initial value that will be added to the input buffer following a spike from a preceding neuron.

Note that the input will have already been delayed by the appropriate amount before it reaches this function, and that the input weights from several spikes may be combined into a single weight. Additionally, the input weights might be either current or conductance as determined by the input type. The synapse type should not perform any conversion of the weights.

The synapse type header file also defines the functions that make up the interface of the synapse type API. The synapse Type API requires the following interface functions to be implemented.

- ```
  static void synapse_types_shape_input(
        input_t *input_buffers, index_t neuron_index,
        synapse_param_t* parameters);
  ```
  Shapes the values (current or conductance) in the input buffers for the synapses of a given neuron. The input buffers for all neurons and synapse types are given here, and the following function can be used to obtain the index of the appropriate input buffer given the indices of the neuron and of the synapse (e.g. if there is an excitatory and inhibitory synapse per neuron, the indices might be 0 and 1 respectively):
  ```
  index_t synapse_types_get_input_buffer_index(synapse_index,
  neuron_index)
  ```

- ```
  static void synapse_types_add_neuron_input(
        input_t *input_buffers, index_t synapse_type_index, index_t
  neuron_index,
        synapse_param_t* parameters, input_t input)
  ```

Adds a synaptic weight input to the input buffer for a given synapse of a given neuron after a spike has been received (and appropriately delayed). This allows the weight to be scaled as required before it is added to the buffer.

- `static input_t synapse_types_get_excitatory_input(`
  `input_t *input_buffers, index_t neuron_index)`
  Returns the total combined excitatory input from the buffers available for a given neuron id. Note that if several synapses are excitatory, this function should add up the input values (or perform an otherwise appropriate function) to return the total excitatory input value.

- `static input_t synapse_types_get_inhibitory_input(`
  `input_t *input_buffers, index_t neuron_index)`
  Extracts the total combined inhibitory input from the buffers available for a given neuron id. Note that if several synapses are inhibitory, this function should add up the input values (or perform an otherwise appropriate function) to return the total inhibitory input value. Note also that the value should be a positive number; subtraction is performed in the neuron model as required.

- `static const char *synapse_types_get_type_char(index_t synapse_type_index)`
  Returns a human readable character for the type of synapse. Examples would be X = excitatory types and I = inhibitory types.

- `static void synapse_types_print_parameters(synapse_param_t *parameters)`
  Prints the static parameters of the synapse type. This is currently only executed when the models are in debug mode.

- `static void synapse_types_print_input(`
  `input_t input_buffers, index_t neuron_index)`
  Prints the input for a neuron id given the available inputs. This is currently only executed when the models are in debug mode.

See synapse_types_my_impl.h for an example of an implementation of a synapse type.

Threshold types

The threshold type header file defines the `threshold_type_t` data structure that declares the parameters required for the threshold type. This might commonly include the actual threshold value amongst other parameters. The header also defines the functions that make up the interface of the threshold type API. The threshold Type API requires the following interface functions to be implemented.

- `static bool threshold_type_is_above_threshold(`
  `state_t value, threshold_type_pointer_t threshold_type)`
  Determines if the threshold has been reached; if the neuron is to spike, given the value of the state variable, true is returned, otherwise false is returned.

Set my_threshold_type.h for an example of an implementation of a threshold type.

Additional inputs

The additional input header file defines the `additional_input_t` data structure, which declares the parameters required for the additional input. The header also defines the functions that make up the interface of the additional input type API. The additional input Type API requires the following interface functions to be implemented:

- `static input_t additional_input_get_input_value_as_current(`
  `additional_input_pointer_t additional_input, state_t membrane_voltage)`

Gets the value of current provided by the additional input. This may or may not be dependent on the membrane voltage.

- `static void additional_input_has_spiked(`
  `additional_input_pointer_t additional_input)`
  Notifies the additional input type that the neuron has spiked.

Python Model Builds

Once the C code has been constructed, the PyNN model must be created in Python to translate the PyNN parameters into a form that the C code can understand. In PyNN, populations can be made up of an arbitrary number of neurons, however to maintain real-time operation the number of neurons that are simulated on each core must be limited. The PACMAN module is used by sPyNNaker to partition the populations into subpopulations, based on the specified maximum number of atoms per core of the model, as well as the resources required by the synaptic matrix. The DataSpecification module is then used to write the data for each subpopulation. This is then loaded onto the machine, along with binary executable, using SpiNNMan.

As with the C code, there are number of components that can be re-used, so that only properties relevant to the new model itself need to be defined. This is done by constructing an individual component for:

1. Neuron model,
2. Input type,
3. Synapse type,
4. Threshold type,
5. Additional input.

These 5 components are then handed over to the main interface object that every neuron model has to extend.

If we look at the my_model_curr_exp in python_models -> neuron -> builds directory, we will see the code shown in **Code 4** where the my_model_curr_exp builds its components and hands them over to the main sPyNNaker interface. The breakdown is as follows:

1. On Lines 91 and 92 the neuron model component is created.
2. On Lines 96 and 97 the synapse type component is created.
3. On Line 101 the input type component is created.
4. On Line 105 the threshold type component is created.
5. Line 109 shows that this model does not contain any additional input components.
6. Lines 113 to 135 show the handing over of these separate components to the sPyNNaker main system which will handle all the python support. Note that the binary must match the name of the aplx file generated by the C code.

```
89 .    # TODO: create your neuron model class (change if required)
90.     # create your neuron model class
91.     neuron_model = MyNeuronModel(
92.       n_neurons, machine_time_step, i_offset, my_parameter)
93.
94.     # TODO: create your synapse type model class (change if required)
95.     # create your synapse type model
96.     synapse_type = SynapseTypeExponential(
97.       n_neurons, machine_time_step, tau_syn_E, tau_syn_I)
98.
99.     # TODO: create your input type model class (change if required)
100.    # create your input type model
101.    input_type = InputTypeCurrent()
```

```
102.
103.    # TODO: create your threshold type model class (change if required)
104.    # create your threshold type model
105.    threshold_type = ThresholdTypeStatic(n_neurons, v_thresh)
103.
107.    # TODO: create your own additional inputs (change if required).
108.    # create your own additional inputs
109.    additional_input = None
110.
111.    # instantiate the sPyNNaker system by initializing
112.    #  the AbstractPopulationVertex
113.    AbstractPopulationVertex.__init__(
114.
115.        # standard inputs, do not need to change.
116.        self, n_neurons=n_neurons, label=label,
117.        machine_time_step=machine_time_step,
118.        timescale_factor=timescale_factor,
119.        spikes_per_second=spikes_per_second,
120.        ring_buffer_sigma=ring_buffer_sigma,
121.        incoming_spike_buffer_size=incoming_spike_buffer_size,
122.
123.        # TODO: Ensure the correct class is used below
124.        max_atoms_per_core=MyModelCurrExp._model_based_max_atoms_per_core,
125.
126.        # These are the various model types
127.        neuron_model=neuron_model, input_type=input_type,
128.        synapse_type=synapse_type, threshold_type=threshold_type,
129.        additional_input=additional_input,
130.
131.        # TODO: Give the model a name (shown in reports)
132.        model_name="MyModelCurrExp",
133.
134.        # TODO: Set this to the matching binary name
135.        binary="my_model_curr_exp.aplx")
```

Code 4: Subsection of the my_model_curr_exp.py class

Take care to note that the same components are used in the python as are used in the c code's *Makefile*. This means for every new component you build for a neuron build in c which is not originally supported by the sPyNNaker tools, you need to build a corresponding python component file.

In the new_template folder there are a set of template files within the template directory for each python component. These are located under python_models -> neuron. These detail the parts of the class that need to be changed for your model.

Python __init__.py files

Most of the __init__.py files in the template do not contain any code. The one within `python_models` is the exception; this file adds the `model_binaries` module to the executable paths, allowing sPyNNaker to search this folder for your compiled binary. You can also import your module here to make it easy to use in other scripts.

Python setup.py file

This file enables you to install the new module. This is set up to install all the modules in the template; if you add any modules, these also need to be added to this file (it is not recursive; each module has to be

added separately). To add the module to your python environment in such a way that you can still edit it, you can run:

```
[sudo] python setup.py develop [--user]
```

You need to use `sudo` if you are installing centrally on Linux or Mac OS X; on windows you need to be in an Administrative console. Add `--user` instead if you want to install only for your username (you shouldn't mix these two options, or you will end up installing it only for the root user).

Using your module

In order to use the new module, you need to import your module in addition to PyNN e.g. for the template module, you can do the following:

```
import pyNN.spiNNaker as p
from python_models.neuron.builds.my_model_curr_exp import MyModelCurrExp
pop = p.Population(1, MyModelCurrExp, {})
```

A more detailed example is shown in the template in `examples/my_example.py`.

Task 1: Simple Neuron Model [Easy]

This task will create a simple neural model using the template, and execute it on SpiNNaker.

1. Change the my_neuron_model_impl.c and .h templates by adding two parameters, one representing a decay and one representing a rest voltage. The parameters should be `REAL` values.
2. Change the model to subtract the difference between the current voltage and the rest voltage multiplied by the decay from the membrane voltage, before adding the total input i.e.

   ```
   v_membrane = v_membrane - ((v_membrane - v_rest) * decay) + input
   ```

3. Recompile the binary.
4. Update the python code model to accept the new decay and rest voltage parameters, ensuring that they match the order of the C code (use `DataType.S1615`). Add getters and setters for the values and update the number of neuron parameters.
5. Update the python code builds to accept the new parameters with default values of 0.1 for decay and -65.0 for the rest voltage.

Run the example script and see what happens.

Task 2: Conductance-based Model [Moderate]

This task will build a conductance-based model.

1. Make a copy of the C build folder for my_model_curr_exp to my_model_cond_exp.
2. Change the Makefile so that it uses the conductance input type and ensure that the binary name is different from the current based model.
3. Build the binary.
4. Copy the python model my_model_curr_exp.py to my_model_cond_exp.py and update the code to use the conductance input type, including adding the new required parameters for conductance, and the binary name and model name.
5. Update the example script to use the new model, adjusting the weights to be conductances (usually much smaller values e.g. 0.1 should be enough)

Run the example script and see what happens.

Task 3: Stochastic Threshold Model [Hard]

This task will create a new threshold model for stochastic thresholds.

1. Update the template threshold type my_threshold_type.c and .h, removing the parameter my_param, and adding a parameter representing the probability of the neuron firing if it is over the threshold value. This will be a `uint32_t` value in C (see later for details).

2. Add another parameter which is the seed of the random number generator. This is an array of 4 `uint32_t` values for the simplest random number generator in `random.h` (from the spinn_common library - as this should have been installed, you can use `#include <random.h>`).

3. Update the threshold calculation so that when the membrane voltage is over the threshold voltage, the RNG is called with the seed (`mars_kiss64_seed(mars_kiss64_seed_t seed)`).

4. Update the threshold calculation to only result in a spike if the value returned from the RNG is greater than than the probability value.

5. Rebuild the my_model_curr_exp_my_threshold_type.

6. Update the my_threshold_type.py python code to include the new parameters, and to generate the random seed. The probability parameter will be between 0 and 1 in Python (default of 0.5), but as the random number generator generates an integer value, this should be converted into a `uint32_t` value between 0 and 0x7FFFFFFF. The seed can be generated using a PyNN RNG, which can be provided to the model as a parameter. Once generated, the seed should be validated using:

   ```
   spynnaker.pyNN.utilities.utility_calls.validate_mars_kiss_64_seed(seed)
   ```

   where `seed` is an array of 4 integer values. Note that `seed` will be updated in place.

7. Update the my_model_curr_exp_my_threshold_type.py build to include the new parameters and pass them in to the threshold type. Make rng an optional parameter, which if not set uses a new NumpyRNG.
8. Update the example script to decrease the threshold value to ensure that the model fires.

Run the example script and see how the number of spikes differs for different settings of the spike probability.

Adding new models of synaptic plasticity

August 28, 2015

Contents of package

examples/stdp_triplet.py PyNN script that reproduces experimental protocol developed by Sjöström et al. [2].

neural_modelling/src/neuron/Makefile Makefile which lists all the neuron models defined in this module.

neural_modelling/src/neuron/builds/Makefile.common Makefile which lists new STDP components defined by this module.

neural_modelling/src/neuron/builds/IF_curr_exp_stdp_mad_pair_additive/Makefile Makefile to build SpiNNaker executable with spike-pair STDP rule.

neural_modelling/src/neuron/builds/IF_curr_exp_stdp_mad_triplet_additive/Makefile Makefile to build SpiNNaker executable with Pfister and Gerstner [1] spike-triplet STDP rule.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_pair_impl.c C source file containing setup code for spike-pair STDP timing dependence.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_pair_impl.h C header file containing implementation of spike-pair STDP timing dependence discussed in presentation.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_triplet_impl.c C source file containing setup code for spike-triplet STDP timing dependence.

neural_modelling/src/neuron/plasticity/stdp/timing_dependence/timing_triplet_impl.h C header file containing implementation of spike-triplet STDP rule discussed in presentation.

workshop_2015_adding_synaptic_plasticity/__init__.py Python module entry point containing code to hook module into sPyNNaker and import timing dependences sub-module.

workshop_2015_adding_synaptic_plasticity/spike_pair_time_dependency.py
Python class to instantiate and configure spike-pair timing dependence from PyNN.

workshop_2015_adding_synaptic_plasticity/spike_triplet_time_dependency.py
Python class to instantiate and configure spike-triplet timing dependence from PyNN.

Additional code changes

My presentation covered the code changes that are required to implement the behaviour spike-triplet rule on SpiNNaker. However there are some other, less interesting changes that are also required to build a functioning learning rule. Remaining changes to Python and C are discussed in comments at the following URL `http://tinyurl.com/ouk2gj2`.

Exercises

These are all more suggestions than anything else, I'd be interested to help with any triplet-rule based experimentation.

Exercise 1

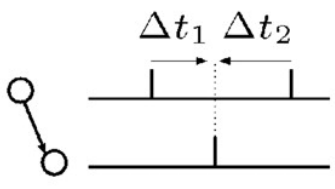
As mentioned in the presentation, the SpiNNaker package already comes with an implementation of the full spike-triplet rule developed by Pfister and Gerstner [1]. This is more computationally expensive than the version developed in this workshop session, but the extra parameters may potentially allow it to better fit experimental data. Try switching the stdp_triplet.py example in the package to use this rule, configured with the parameters fitted by Pfister and Gerstner:

```
timing_dependence = sim.PfisterSpikeTripletRule(
        tau_plus=16.8, tau_minus=33.7,
        tau_x=101, tau_y=114)

weight_dependence = sim.AdditiveWeightDependence(
        w_min=0.0, w_max=max_weight,
        A_plus=5E-10 * start_w, A_minus=7E-3 * start_w,
        A3_plus=6.2e-3 * start_w, A3_minus=2.3E-4 * start_w)
```

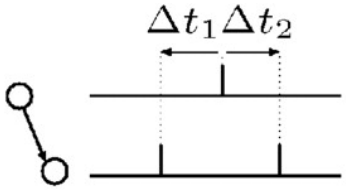
Does this actually reduce the error compared to the version developed in this workshop? Why might this be? The talk this morning on 'Maths & fixed point libraries' may give you some clues!

Exercise 2

Pfister and Gerstner [1] also fitted their model to some experimental data by Wang et al. [3]. These follow the spike-triplet protocol shown in figure 1 which



(a) Pre-post-pre triplet



(b) Post-pre-post triplet

Figure 1: Wang et al. [3] triplet protocol. Each experiment consists of 60 triplets of spikes, one second apart.

Δw	Δt_1	Δt_2
-0.01 ± 0.04	5	-5
0.03 ± 0.04	10	-10
0.01 ± 0.03	15	-5
0.24 ± 0.06	5	-15

(a) Pre-post-pre triplets

Δw	Δt_1	Δt_2
0.33 ± 0.04	-5	5
0.34 ± 0.04	-10	10
0.22 ± 0.08	-15	-5
0.29 ± 0.05	-5	15

(b) Post-pre-post triple

Table 1: Weight changes induced by Wang et al. [3] triplet protocol.

resulted in the weight changes shown in table 1. Can you make a version of stdp_triplet.py that reproduces this protocol?

References

[1] Jean-Pascal Pfister and Wulfram Gerstner. Triplets of spikes in a model of spike timing-dependent plasticity. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 26(38):9673–82, September 2006. ISSN 1529-2401. doi: 10.1523/JNEUROSCI.1425-06.2006. URL `http://www.ncbi.nlm.nih.gov/pubmed/16988038`.

[2] P J Sjöström, G G Turrigiano, and S B Nelson. Rate, timing, and cooperativity jointly determine cortical synaptic plasticity. *Neuron*, 32(6):1149–64, December 2001. ISSN 0896-6273. URL `http://www.ncbi.nlm.nih.gov/pubmed/11754844`.

[3] Huai-Xing Wang, Richard C Gerkin, David W Nauen, and Guo-Qiang Bi. Coactivation and timing-dependent integration of synaptic potentiation and depression. *Nature neuroscience*, 8(2):187–93, February 2005. ISSN 1097-6256. doi: 10.1038/nn1387. URL `http://www.ncbi.nlm.nih.gov/pubmed/15657596`.