Report Week 2

Achievements:
- Set up repository for writing graph_front_end code on https://github.com/AlexRiepenhausen/Number_Crunch, which essentially the repository where all of my final year project code will go
- Python parser for reading data files in csv format built
- Participated in parts of the SpiNNaker workshop

What I learned:
After the workshop on Tuesday, I had the SpiNNaker development team (mostly Alan) giving me a hello_world demo walk-through.

In hello_world.py:
```
import spinnaker_graph_front_end as front_end
front_end.run(x)
if x is 0, the simulation will run indefinitely
else x is the number of milliseconds the simulation will be running for
```

Before that, all relevant vertices (and later edges) have to added to the simulation via invoking front_end.add_machine_vertex

Once the simulation has stopped running, there are two ways of retrieving relevant information regarding the execution:
1. hello_world = placement.vertex.read(placement, buffer_manager)
2. front_end.stop()

1. reads the data directly from SDRAM once execution has finished, which requires the core to put the data back to SDRAM into the relevant partitions (up to 16) after processing (more on that later)

2. On terminating the simulation, the iobuf section within sarc within the SDRAM is extracted. That can be used to retrieve log information for debugging puproses. However, the size of iobuf is very limited (currently 8Mbytes according to my knowledge)

In hello_world_vertex.py
```
DATA_REGIONS = Enum (
  value="DATA_REGIONS",
  names=[ ("SYSTEM", 0),
          ("STRING_DATA", 1) ] )
```

Here, the SDRAM is effectively partitioned into two reagions (not counting the header): The system region with id zero and the string data region with id 1

However, since there is only one data region, when referenced the id 0 is used to indicate that this is the first (and only) region for holding data. The maximum number of partitions is apparently 16

```
def generate_machine_data_specification()

  self._reserve_memory_regions(spec, setup_size)

  spec.switch_write_focus(self.DATA_REGIONS.SYSTEM.value)
  spec.write_array(…)

  spec.switch_write_focus(self.DATA_REGIONS.STRING_DATA.value)
  spec.write_array(…)

  spec.end_specification()
```

This method initialises the two SDRAM regions by performing a switch on the variable “spec” and then using the write_array method
Previously, memory on the SDRAM needs to be reserved with _reserve_memory_regions

```
def read(self, placement, buffer_manager):
```

placement → location of vertex specified by x,y,p with x and y indicating the chip id and p indicating the id of the chip's core

```
  data_pointer, missing_data = buffer_manager.get_data_for_vertex(placement, 0)
```

The buffer_manager is responsible for managing any recording region. The data_pointer points to a byte array that turns into a string on output

```
  record_raw = data_pointer.read_all()
  output = str(record_raw);
```

This particular bit of code then links to the C file

In hello_world.c

```
void c_main()

  //…

 uint32_t timer_period;
 initialize(&timer_period)
```

→ initialize takes the timer (in this case 10) provide by front_end.run(10) in hello_world.py. The timer is also referred to as simulation tick.

```
static bool initialize(uint32_t *timer_period)

  address_t address = data_specification_get_data_address();
  data_specification_read_header(address)
```

data_specification_get_data_address() retrieves the memory address of the header in this case, which is then used to retrieve the addresses of the partitions (also known as DSG-Data Specification Generator regions):

```
simulation_initialise(
  data_specification_get_region(SYSTEM_REGION, address),
  …,&simulation_ticks, SDP, DMA)
```

data_specification_get_region(SYSTEM_REGION, address) takes the address of the header and retrieves the address of the system region within the SDRAM

The next initalization function:

```
static bool initialise_recording()
  address_t address = data_specification_get_data_address();
  address_t recording_region =
    data_specification_get_region(RECORDED_DATA, address);
```

Same story as above, just with a differend region

```
bool success = recording_initialize(recording_region,…)
```

Furthermore, c_main sets the timer tick with spin1_set_timer_tick(timer_period)
Events are handled with two types of callbacks:

```
spin1_callback_on(MCPL_PACKET_RECEIVED, receive_data, MC_PACKET)
spin1_callback_on(TIMER_TICK, update, TIMER)
```

The first callback is invoked whenever the core responsible for executing the code receive a spike (or packet), whereas the second one runs by default until the timer hits a user specified threshold. No packets are exchanged in this particular example program, but it is important to note that synchronisation between cores might become an issue when running large applications (e.g. when certain cores are flooded with callbacks while other cores remain idle).

update → handles the timer and stops application whenever certain conditions are fulfilled

receive_data → basically a stub in this program, since no packets are exchanged in hello_world

simulation_run() → runs simulation

Problems
- Coursework demands more and more attention
- Cannot install the gcc-arm-none-eabi compiler on the university machine, since together with the spiNNaker related software this would exceed my 2GB quota. I requested IT services to raise my quota; this may take some time and possibly requires authorization from my superviser (aka Steve Furber).

Goals for next week:
- finally implement a simple SpiNNaker compatible program based upon given examples
- potentially create a working function to derive a frequency histogram from csv data