Head direction network

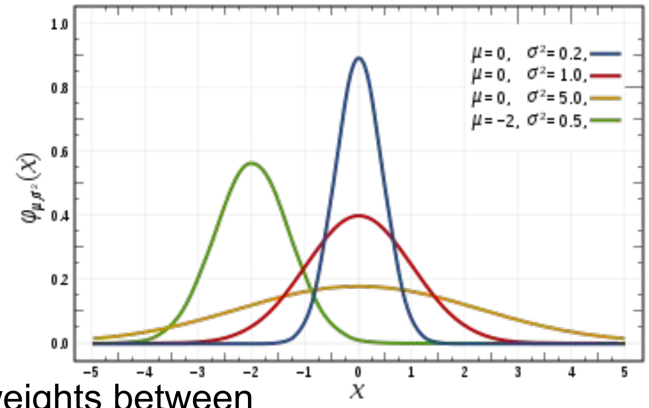
Parameters

N_ex = 180
N_in = N_ex
N_cj = N_ex

Number of cell in each population

sigma = 0.12
mu = 0.5
delay = 0.1
base_ex = 4000
base_in = 450
base_cj = 169
w_ex_cj = 660

Parameters for describing connection weights between populations



I_init = 300.0 #pA
I_init_dur = 100.0 #ms
I_init_pos = N_ex//2

size , duration and excitatory cell number to initialize the bump

Create populations

```
exc = sim.Create("iaf_psc_alpha",N_ex, params={"I_e": 450.})
inh = sim.Create("iaf_psc_alpha",N_in)

l = sim.Create("iaf_psc_alpha",N_cj)
r = sim.Create("iaf_psc_alpha",N_cj)
```

Continuous current input to excitatory population produces spontaneous firing which keeps the bump going

One of many default cell types described in NEST, this is the basic cell type used in example networks etc

Basic leaky integrate and fire neuron

Creating the bump (connections between excitatory and inhibitory populations)

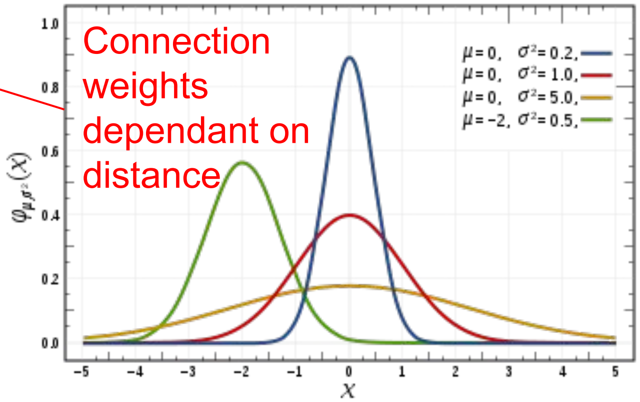
```
w_ex = np.empty((N_in,N_ex))
w_in = np.empty((N_ex,N_in))
for e in range(N_ex):
    for i in range(N_in):
        d1 = abs(e/N_ex - i/N_in)
        d2 = abs(e/N_ex - i/N_in -1)
        d3 = abs(e/N_ex - i/N_in +1)
        d = min(abs(d1),abs(d2),abs(d3))
        w_gauss = np.exp(-(d)**2/2/sigma**2)
        w_ring = np.exp(-(d - mu)**2/2/sigma**2)
        w_ex[i,e] = base_ex * w_gauss
        w_in[e,i] = base_in * w_ring

w_ex[w_ex<10]=0
w_in[w_in<10]=0
```

Cycle through all pairs of excitatory and inhibitory cells

Find distance between the excitatory and inhibitory cell also looking both ways around the ring

Using the smallest distance (magnitude only)

Connection weights dependant on distance



Mu term offsets the gaussian -> strong weights either side of the equivalent excitatory cell

Set small values to 0

Creating the bump

```
bump_init = sim.Create('step_current_generator', 1, params = {'amplitude_times':[0.1,0.1+I_init_dur],
                                                              'amplitude_values':[I_init,0.0]})
sim.Connect(bump_init,[exc[I_init_pos]])
```

Create a step_current_generator device which steps up to 300pA at time 0.1 and down 100ms later

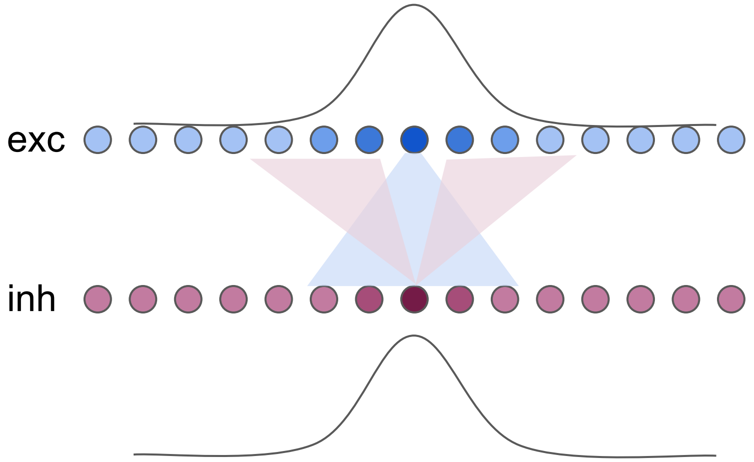
Connect this to a cell in the exc population which becomes the initial center of the bump

Note: sim.connect requires a list of cell numbers even if it is a single element list

Simple HD circuit using Excitatory-inhibitory attractor network

Excitatory cells fire spontaneously

More energy is injected into one cell to initialize the bump



Inhibitory limits the spontaneous activity of the excitatory cells

At the bump center inhibitory input is lowest, so excitatory cells in the bump location continue to fire

Ideothetic input (weights from conjunctive cells to exc population)

```
w_l = np.empty((N_ex,N_cj))
w_r = np.empty((N_ex,N_cj))
for c in range(N_cj):
    for e in range(N_ex):
        d1 = abs((e-1)/N_cj - c/N_ex)
        d2 = abs((e-1)/N_cj - c/N_ex -1)
        d3 = abs((e-1)/N_cj - c/N_ex +1)
        d = min(abs(d1),abs(d2),abs(d3))
        w_l[e,c] = base_cj * (np.exp(-(d)**2/2/sigma**2))

        d1 = abs((e+1)/N_cj - c/N_ex)
        d2 = abs((e+1)/N_cj - c/N_ex -1)
        d3 = abs((e+1)/N_cj - c/N_ex +1)
        d = min(abs(d1),abs(d2),abs(d3))
        w_r[e,c] = base_cj * (np.exp(-(d)**2/2/sigma**2))

m = np.amax(w_l)
w_l[w_l<m] = 0
m = np.amax(w_r)
w_r[w_r<m] = 0
```

Honestly this is legacy code and could be implemented a lot simpler. Currently calculates weight based on distance like the exc-inh pops

Here the excitatory cell is offset by +1 or -1 around the ring when calculating the distance

Then just the peak connection weight is maintained

Each cell is connected to one cell one step CW or ACW around the ring

Need to specify weight 0 for all the other cell pairs - still need full weight matrix

Ideothetic input (AHV from head_angle.csv or head_pose.csv)

	Time	X	Y	Theta	Simulation_reset
0	0.02	0.100000	0.000000e+00	0.000000e+00	NaN
1	0.04	0.010429	-8.580169e-10	-8.227311e-08	NaN
2	0.06	0.016648	1.523187e-06	-4.601878e-05	NaN
3	0.08	0.032111	2.386036e-06	-6.189506e-05	NaN
4	0.10	0.045385	3.045344e-06	-7.203980e-05	NaN
...	...	...	...	...	...

Vel is the difference between theta samples

Positive values are funneled to go_l and negative to go_r

step_current _generator used again to set the current to these 'Ivel' values at each timestep

Current input is sent to all conj cells

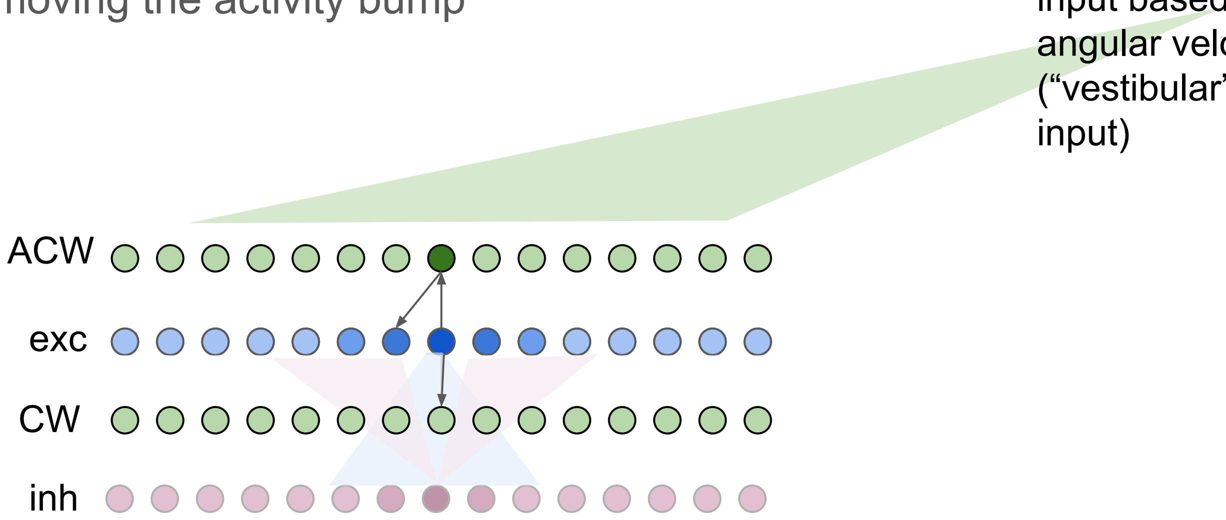
These files saved by the transfer function have a regular 50Hz sample rate

Time must be converted to ms

```
Ivel = vel * 0.35 * 10000

sh = 150
go_l,go_r = Ivel,-Ivel
go_l = go_l+sh
go_r = go_r+sh
go_l[go_l<=sh] = 0
go_r[go_r<=sh] = 0

# Connect AV input to conjunctive layers
l_input = sim.Create('step_current_generator', 1)
sim.SetStatus(l_input,{'amplitude_times': t[1:],'amplitude_values': go_l})
r_input = sim.Create('step_current_generator', 1)
sim.SetStatus(r_input,{'amplitude_times': t[1:],'amplitude_values': go_r})

sim.Connect(r_input,r,'all_to_all')
sim.Connect(l_input,l,'all_to_all')
```

Simple HD circuit - moving the activity bump

Go anti- clockwise input based on head angular velocity (“vestibular”/odometry input)



Connecting populations

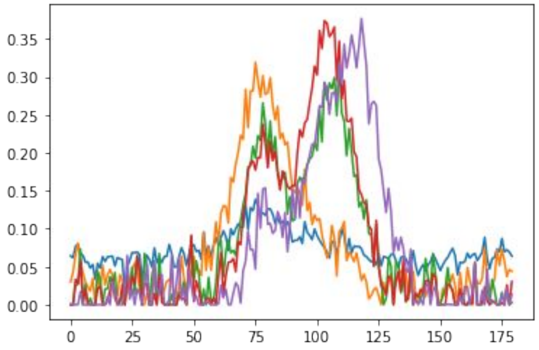
```
exc_2_inh = sim.Connect(exc,inh,'all_to_all',syn_spec={'weight': w_ex, 'delay': delay})
inh_2_exc = sim.Connect(inh,exc,'all_to_all',syn_spec={'weight': -w_in, 'delay': delay})

l_2_exc = sim.Connect(l,exc,'all_to_all',syn_spec={'weight': w_l, 'delay': delay})
r_2_exc = sim.Connect(r,exc,'all_to_all',syn_spec={'weight': w_r, 'delay': delay})
```

Weight matrices define connections between each pair of cells in the two populaitons

```
exc_2_l = sim.Connect(exc,l,'one_to_one',syn_spec={'weight': w_ex_cj, 'delay': delay})
exc_2_r = sim.Connect(exc,r,'one_to_one',syn_spec={'weight': w_ex_cj, 'delay': delay})
```

Each excitatory cell is connected to its equivalent conj cell one to one with a set weight

Allothetic input



Using reconstructions_head_direction.npy and body_pose.npy

```
predNet = np.load('reconstructions_head_direction.npy')
predtm = np.load('body_pose.npy')

predNet[predNet<0] = 0
predNet[predNet>0] = predNet[predNet>0]*1.
predNet = np.vstack([predNet,np.zeros([1,180])])

predidx = np.array(predtm[:,0]).astype(int)
predtm = np.take(pos_t, predidx)
```

Make sure predictions are all positive + option to scale predicitons

Get relative index each prediction occurs at, and find the time each prediction occurred

```
view_input = sim.Create('step_current_generator', N_ex)
for n in range(N_ex):
    sim.SetStatus([view_input[n]],{'amplitude_times': prediciton_times,'amplitude_values': predicitons[:,n]})

sim.Connect(view_input,exc,'one_to_one')
```

One step current generator per excitatory cell, use the value of the prediction at each time step as current input to each cell

Recording spikes

```
sim.SetKernelStatus({"overwrite_files": True,"data_path": folder,"data_prefix": txt})
```

Set the folder and data prefix if saving spikes to file

```
exc_spikes = sim.Create("spike_detector", 1, params={"withgid": True,"withtime": True, "to_file": True, "label": "gc_spikes"})
sim.Connect(exc,exc_spikes)
```

Can also just store in variable

```
exc_spikes = sim.Create("spike_detector", 1, params={"withgid": True,"withtime": True})
sim.Connect(exc,exc_spikes)
```

Run simulation

```
sim.ResetKernel()
```

Must reset kernel between every new simulation

Either all at once

```
tic = tm.time()
sim.Simulate(sim_len)
print(f'Simulation run time: {np.around(tm.time()-tic,2)} s  Simulated time: {np.around(sim_len/1000,2)} s')
```

```
Simulation run time: 132.58 s  Simulated time: 180.0 s
```

Or in chunks

```
sim.Prepare()
for i in np.arange(sim_len/chunk):
    sim.Run(chunk)
    print(f'Chunk {i} complete...')
sim.Cleanup()
```

Finding the center of the bump

```
ev = sim.GetStatus(exc_spikes)[0]['events']
t = ev['times']
sp = ev['senders']

dt = 20
T = np.arange(0,(len(theta)*dt),dt*2)

modes = np.zeros(len(T))
modes[:] = np.nan
rates = np.zeros((N_ex,len(time)))
for i in range(len(T)-1):
    idx = (t>T[i])*(t<T[i+1])
    lst = sp[np.where(idx)]
    occurence_count = Counter(lst)
    active = occurence_count.keys()
    for cell in active:
        rates[cell-1,i] = occurence_count[cell]
    mode = occurence_count.most_common(1)
    if len(mode):
        modes[i] = mode[0][0]

step = (2*np.pi)/N_ex
modes = (modes*step) - np.pi
```

For spikes stores in variable

40ms bins

Find all spikes in the bin

Find the most active cell

Find the most active cell

Convert to radians

```
for file in os.listdir(folder):
    if file.endswith(".gdf"):
        data = pd.read_csv(f'{folder}/{file}', delimiter="\t")
        data = data.drop(columns=['Unnamed: 2'])
        data = data.set_axis(['sp','t'], axis=1, inplace=False)
        sp = np.array(data['sp'])
        tms = np.array(data['t'])
```

Read spike data in from file

Grid cell network

Only including parts that are different

Parameters

```
y_dim = (0.5* np.sqrt(3))
Nx = 20
Ny = int(np.ceil(Nx * y_dim))
N = Nx * Ny
```

Y_dim is a parameter used to produce the twisted torus connectivity described in Guanella et al 2007

Cells are arranged in a sheet x by y cells

Populations

```
exc = sim.Create("iaf_psc_alpha",N, params={"I_e": 400.})
inh = sim.Create("iaf_psc_alpha",N)

l = sim.Create("iaf_psc_alpha",N)
r = sim.Create("iaf_psc_alpha",N)
u = sim.Create("iaf_psc_alpha",N)
d = sim.Create("iaf_psc_alpha",N)
```

Four conj populations this time to traverse the 2D sheed rather than 1D ring

Creating the bump (connections between excitatory and inhibitory populations)

```
w_ex = np.empty((N,N))
w_in = np.empty((N,N))
for e in range(N):
    x_e = (e%Nx) / Nx
    y_e = y_dim*(e//Nx)/ Ny
    for i in range(N):
        x_i = (i%Nx) / Nx
        y_i = y_dim*(i//Nx) / Ny
```

Cycle through all pairs of excitatory and inhibitory cells finding the xy position of each cell in the sheet

```
        d1 = np.sqrt(abs(x_e - x_i)**2 + abs(y_e - y_i)**2)
        d2 = np.sqrt(abs(x_e - x_i - 0.5)**2 + abs(y_e - y_i + y_dim)**2)
        d3 = np.sqrt(abs(x_e - x_i - 0.5)**2 + abs(y_e - y_i - y_dim)**2)
        d4 = np.sqrt(abs(x_e - x_i + 0.5)**2 + abs(y_e - y_i + y_dim)**2)
        d5 = np.sqrt(abs(x_e - x_i + 0.5)**2 + abs(y_e - y_i - y_dim)**2)
        d6 = np.sqrt(abs(x_e - x_i - 1.)**2 + abs(y_e - y_i)**2)
        d7 = np.sqrt(abs(x_e - x_i + 1.)**2 + abs(y_e - y_i)**2)
```

Find distance between the excitatory and inhibitory cell
See Guanella et al 2007 for full description of calculating the distances between cells in the twisted torus

```
        d_ = min(d1,d2,d3,d4,d5,d6,d7)
```

Using the smallest distance (magnitude only)

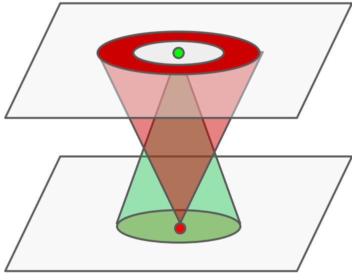
```
        w_gauss = np.exp(-(d_)**2/2/sigma**2)
        w_ring = np.exp(-(d_ - mu)**2/2/sigma**2)
```

Connection weights dependant on distance (gaussian function)

Mu term offsets the gaussian -> strong weights in a ring around equivalent excitatory cell

```
        w_ex[i,e] = base_ex * w_ring
        w_in[e,i] = base_in * w_gauss

w_ex[w_ex<10]=0
w_in[w_in<10]=0
```

Set small values to 0



Ideothetic input (weights from conjunctive cells to exc population)

```
w_l = np.empty((N,N))
w_r = np.empty((N,N))
w_u = np.empty((N,N))
w_d = np.empty((N,N))
for e in range(N):
    x_e = (e%Nx) / Nx
    y_e = (e//Nx) / Ny * y_dim
    for i in range(N):
        x_i = ((i%Nx) / Nx) - (1/Nx)
        y_i = (i//Nx) / Ny * y_dim

        d1 = np.sqrt(abs(x_e - x_i)**2 + abs(y_e - y_i)**2)
        d2 = np.sqrt(abs(x_e - x_i - 1.)**2 + abs(y_e - y_i)**2)
        d3 = np.sqrt(abs(x_e - x_i + 1.)**2 + abs(y_e - y_i)**2)
        d4 = np.sqrt(abs(x_e - x_i + 0.5)**2 + abs(y_e - y_i - y_dim)**2)
        d5 = np.sqrt(abs(x_e - x_i - 0.5)**2 + abs(y_e - y_i - y_dim)**2)
        d6 = np.sqrt(abs(x_e - x_i + 0.5)**2 + abs(y_e - y_i + y_dim)**2)
        d7 = np.sqrt(abs(x_e - x_i - 0.5)**2 + abs(y_e - y_i + y_dim)**2)
        d_ = min(d1,d2,d3,d4,d5,d6,d7)
        w_l[i,e] = base_cj * (np.exp(-(d_)**2/2/sigma**2))

        x_i = ((i%Nx) / Nx) + (1/Nx)
        y_i = (i//Nx) / Ny * y_dim

        d1 = np.sqrt(abs(x_e - x_i)**2 + abs(y_e - y_i)**2)
        d2 = np.sqrt(abs(x_e - x_i - 1.)**2 + abs(y_e - y_i)**2)
        d3 = np.sqrt(abs(x_e - x_i + 1.)**2 + abs(y_e - y_i)**2)
        d4 = np.sqrt(abs(x_e - x_i + 0.5)**2 + abs(y_e - y_i - y_dim)**2)
```

The twisted torus is needed to calculate the connections from the conjunctive layers to the exc cells.

Same calculation as exc to inh but take the max weights only

```
m = np.amax(w_l)
w_l[w_l<m] = 0
m = np.amax(w_r)
w_r[w_r<m] = 0
m = np.amax(w_u)
w_u[w_u<m] = 0
m = np.amax(w_d)
w_d[w_d<m] = 0
```

Ideothetic input (velocity samples at 50hz)

```
vel_x = np.diff(pos_x)
vel_y = np.diff(pos_y)

vel_x,vel_y = vel_x*gain, vel_y*gain

go_l,go_r = vel_x,-vel_x
go_u,go_d = vel_y,-vel_y
go_l, go_r, go_u, go_d = go_l+sh, go_r+sh, go_u+sh, go_d+sh
go_l[go_l<=sh] = 0.
go_r[go_r<=sh] = 0.
go_u[go_u<=sh] = 0.
go_d[go_d<=sh] = 0.

l_input = sim.Create('step_current_generator', 1)
sim.SetStatus(l_input,{'amplitude_times': t[1:],'amplitude_values': go_l})
r_input = sim.Create('step_current_generator', 1)
sim.SetStatus(r_input,{'amplitude_times': t[1:],'amplitude_values': go_r})
u_input = sim.Create('step_current_generator', 1)
sim.SetStatus(u_input,{'amplitude_times': t[1:],'amplitude_values': go_u})
d_input = sim.Create('step_current_generator', 1)
sim.SetStatus(d_input,{'amplitude_times': t[1:],'amplitude_values': go_d})

sim.Connect(l_input,l,'all_to_all')
sim.Connect(r_input,r,'all_to_all')
sim.Connect(u_input,d,'all_to_all')
sim.Connect(d_input,u,'all_to_all')
```

Posx posy and Time from datafile

Scale to match appropriate current values

Positive x values are funneled to go_l and negative x to go_r

Positive y values are funneled to go_u and negative y to go_d

step_current _generator used again to set the current to these ‘lvel’ values at each timestep

Current input is sent to all conj cells

Plotting spikes on trajectory

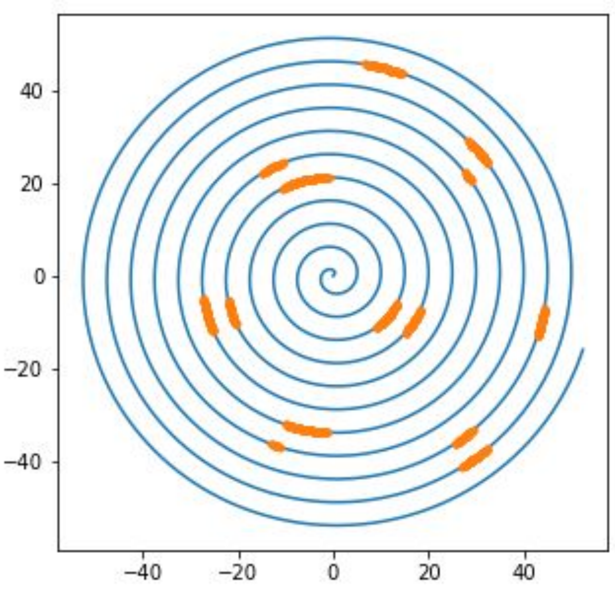
```
occurence_count = Counter(sp)
# print(occurence_count)
cell = occurence_count.most_common(5)[0][0]
print(cell)

spktms = tms[sp==cell]
spktms = (spktms//20)*20
spktms=spktms[1:]

xs = np.empty((len(spktms)))
ys = np.empty((len(spktms)))

for i,spk in enumerate(spktms):
    try:
        xs[i] = pos_x[np.where(time == spk)[0][0]]
        ys[i] = pos_y[np.where(time == spk)[0][0]]
    except:
        a = 1

fig = plt.figure(figsize=(10, 10),facecolor='w')
plt.plot(pos_x,pos_y)
plt.plot(xs,ys,'.')

plt.savefig(f'{folder}/spikes.png')
```

Plot cell that spikes the most

Find spike times of that cell

Find position at each spike time

Plot spikes at these positions on the trajectory



Grid analysis (uses code from Nolan group in Edinburgh available on GitHub)

```
prm = dict()
prm['pixel_ratio'] = 440
prm['output_path'] = f'{folder}/gridAnalysis'
```

```
spike_data = pd.DataFrame()

cells = [cell]
for cell in cells: #set(sp):
    spktms = t[sp==cell]
    spktms = (spktms//20)*20

    xs = np.empty((len(spktms)))
    ys = np.empty((len(spktms)))

    for i,spk in enumerate(spktms):
        if spk < 60000:
            xs[i] = pos_x[np.where(time == spk)[0][0]]
            ys[i] = pos_y[np.where(time == spk)[0][0]]

    spike_data = spike_data.append({'cell_id': int(cell),
                                    'spike_times': spktms,
                                    'number_of_spikes': len(spktms),
                                    'mean_firing_rate': len(spktms)/(sim_len/1000),
                                    'position_x': xs - min(pos_x),
                                    'position_y': ys - min(pos_y),
                                    'position_x_pixels' : (xs - min(pos_x)) / 100 * prm['pixel_ratio'],
                                    'position_y_pixels' : (ys - min(pos_y)) / 100 * prm['pixel_ratio']
                                    }, ignore_index=True)

spike_data.to_pickle(f'{folder}/{file}.pkl')
```

```
spatial_data = dict()
spatial_data['position_x'] = pos_x - min(pos_x)
spatial_data['position_y'] = pos_y - min(pos_y)

# pixels = cm / 100 * prm['pixel_ratio']
spatial_data['position_x_pixels'] = spatial_data['position_x'] / 100 * prm['pixel_ratio']
spatial_data['position_y_pixels'] = spatial_data['position_y'] / 100 * prm['pixel_ratio']

spatial_data = pd.DataFrame.from_dict(spatial_data)
```

Requires data to be arranged in a very specific format

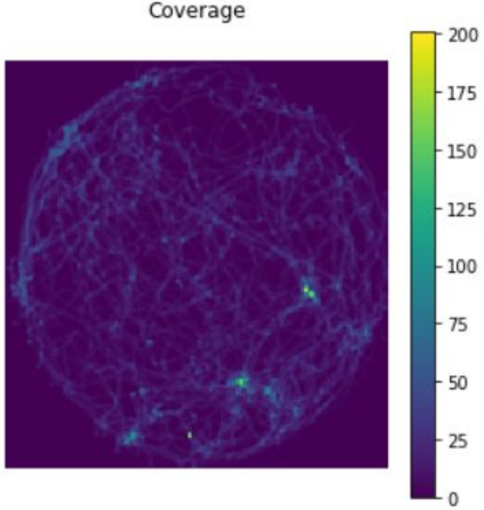
Grid analysis (uses code from Nolan group in Edinburgh available on GitHub)

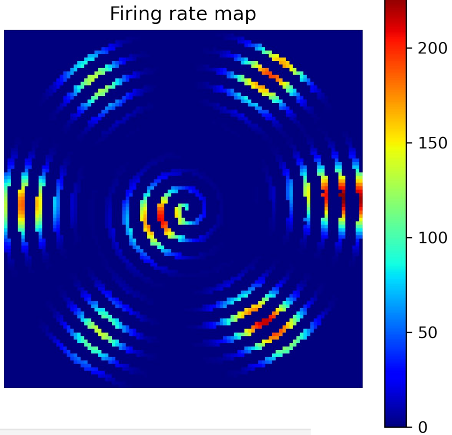
```
import grid_analysis as griddy
```

Ive put all of the adapted functions in a single file

```
position_heat_map = griddy.get_position_heatmap(spatial_data, prm)
griddy.plot_coverage(position_heat_map, prm)
```

```
I will plot a heat map of the position of the animal to show coverage.
```






```
if not 'firing_maps' in spike_data:
    position_heat_map, spike_data = griddy.make_firing_field_maps(spatial_data, spike_data, prm)

spike_data.to_pickle(f'{folder}/{file}_withfiringMaps.pkl')
```

```
# plot firing rate maps
griddy.plot_firing_rate_maps(spike_data, prm)
```

```
spike_data = griddy.process_grid_data(spike_data)
spike_data.to_pickle(f'{folder}/{file}_withGridMetrics.pkl')
```