Investigation of Hippocampus Grid Cells in 3D space

A Project Report

submitted by

TRIVIKRAM THIRUKKONDA

EE14B062

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2018

REPORT CERTIFICATE

This is to certify that the report titled **Investigation of Hippocampus Grid Cells in 3D space**, submitted by **Trivikram Thirukkonda**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Srinivasa Chakravarthy
Project Guide
Professor
Dept. of Biotechnology
IIT-Madras, 600 036

Place: Chennai
Date: 11th May 2018

ACKNOWLEDGEMENTS

This project has been an eye-opening experience into the field of Biotech research. I was exposed to the latest ground-breaking theories and its practical applications. I was given a sandboxed environment from which to experience the cutting-edge of research.

For this, I would like to thank my guide Srinivasa Chakravarthy for giving me an opportunity to work in this field and guiding me throughout the project. I would like to thank Karthik Soman, a Ph.D scholar of the Biotech department for mentoring me.

I would like to thank both the Electrical and Biotech departments for giving me the flexibility to pursue this project as part of my course curriculum.

Finally, I would like to thank all my friends who helped me complete this project successfully.

Trivikram Thirukkonda
EE14B062
Student
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai
Date: 11th May 2018

TABLE OF CONTENTS

Contents

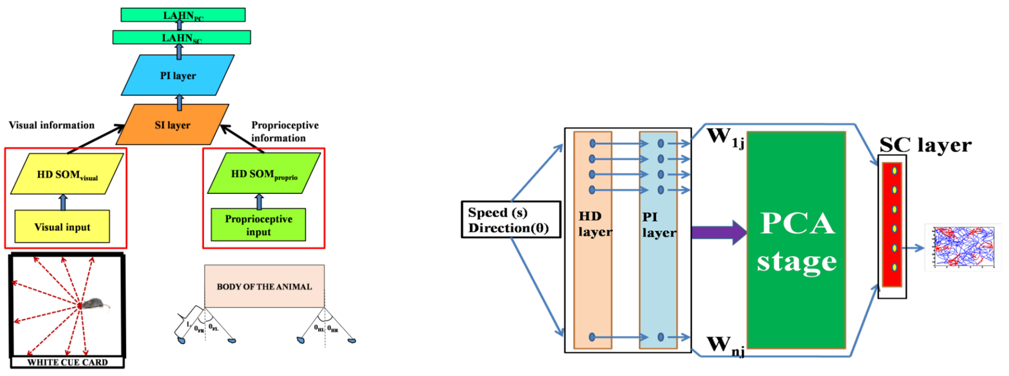
1 Introduction

Navigating the environment is an essential part of an animal's survival and accurately estimating its location in the environment is central to this process. The animal uses sensory cues like vision and proprioception (feedback from movement of limbs) to both familiarize itself with the environment and locate itself in it. When experiments were carried out on animals to investigate the neural architecture responsible for this, researchers found a hierarchy of neurons in the **Hippocampus** region of the brain that seemed to respond to the animal's position, speed and direction.

Further experiments done on animals navigating simple mazes revealed the way the neurons in this hierarchical order are connected. Inputs from visual and proprioceptive neurons feed into sets of neurons called **Self Organising Maps** (SOMs) which act as encoders to map the input onto a finite set of values. These two types of encoded inputs are then combined in the **Sensory Integration** (SI) layer using a weighted average which depends on the reliability of incoming information. If there is enough light for reliable visual information, the visual SOM's value will dominate, and vice-versa. At this point, the information in the SI layer's neurons have information on the direction and velocity of the animal(Soman et al. [4]).

The SI layer feeds into the **Path Integration** (PI) layer via one-to-one connections. As the name suggests, these neurons integrate the incoming velocity information, encoded in the form of phase difference w.r.t theta oscillations (constant, low frequency oscillations independent of Hippocampus). Finally, the PI layer feeds into a **Lateral Anti-Hebbian Network** (LAHN) of neurons in a fully-connected fashion. These neurons encode the incoming information in a manner similar to Principal Component Analysis. Ultimately, the LAHN neurons help the animal locate itself as these neurons fire only when the animal is in certain locations in the environment i.e. the firing patterns have relevant spatial information(Soman et al. [4]). These "spatial cells" are the focus of this study.

Study of the firing patterns of these LAHN neurons in different environments can help us better understand the way in which animals navigate their environments. This could ultimately help us improve artificial techniques of navigation in closed, small environments where classical techniques like GPS are not feasible.



(a) Full model (b) Simplified model

Figure 1: Neural hierarchy under consideration

1.1 Simplified Neural Architecture

The main focus of this project is to investigate the firing field patterns of the LAHN cells. Hence, we do not focus on the methods of acquiring inputs being given to the network. The speed and direction of the animal is directly given as an input to the Head Direction layer instead of acquiring it using vision and proprioception. See fig. 1.

1.1.1 Head Direction Cells

The Head Direction (HD) layer consists of neurons whose response depends on the projection of the animal's direction on their preferred direction.

$$\mathrm{HD_i} = \cos(\theta - \theta_\mathrm{i}) \tag{1}$$

These HD neurons are of two types - azimuthal and pitch neurons. For animals navigating a 2D environment, the pitch neurons are non-existent. Even in animals such as birds, the percentage of pitch neurons is less than azimuthal since the pitch distribution is not as uniform as azimuthal (i.e. the animal doesnt undergo sharp ascents or descents). In the experiments conducted on bats, it was found that pitch and azimuthal neurons were 30% and 70% of the total neurons.

1.1.2 Path Integration Layer

Using the input from HD cells and the supplied velocity information, the PI cells act as a dead reckoning system. There is a one-to-one mapping from the HD cells to the PI cells. Each PI cell, under zero input, undergoes a constant oscillation at a given base frequency. The output of HD cells modulates the frequency of these oscillations and thus the phase of the oscillations encode for the integrated HD cell output.

$$\mathrm{PI_i} = \sin[\int 2\pi(\mathrm{f_0} + \beta \mathrm{sHD_i})\mathrm{dt}] \tag{2}$$

where β is the spatial scaling parameter and $\mathrm{HD_i}$ is the corresponding HD cell output.
These outputs are then thresholded using the following rule:

$$\mathrm{PI_i^{Thr}} = \mathrm{H}(\mathrm{PI_i} - \epsilon_{\mathrm{PI}}).\mathrm{PI_i} \tag{3}$$

where H() is the Heaviside function and ϵ is the threshold. This thresholding is to simulate the neuron's continuous firing threshold.
Finally, these outputs are connected to a single LAHN (as opposed to two LAHNs in the full network model) in a fully connected fashion.

1.1.3 Lateral Anti-Hebbian Networks

The LAHN is a set of neurons connected to each other such that incoming inputs are connected in a Hebbian manner (forward weights) and connections between neurons are connected in an anti-Hebbian manner (lateral weights). The anti-Hebbian connections act as a decorrelation network, removing correlations between incoming inputs as far as possible (P.Foldiak 1989 [5]). As a result, the LAHN acts as an effective dimensionality reduction network while maintaining maximum mutual information flow between input and output. Similar to PCA, this network projects the input onto a subspace of its largest principal components (dimensions with greatest variance) having least cross-correlation between them.

Training this network happens in an unsupervised manner, using simple localized rules for modification of connection weights. Not only is the training faster this way, it is also biologically more plausible than classical error propagation rules. The network training is said to have converged when the maximum change in weight of a connection (both forward and lateral) is less than a specified threshold.

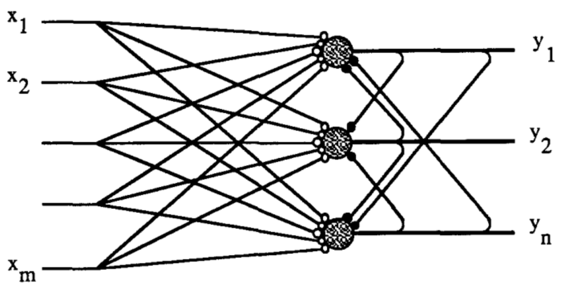


Figure 2: A sample LAHN. White circles - Hebbian connections. Black circles - anti Hebbian

The output of each neuron is as follows:

$$y_i = \sum_{j=1}^{m} q_{ij}x_j + \sum_{j=1}^{n} w_{ij}y_j \tag{4}$$

where w_{ij} and q_{ij} are forward and lateral weights respectively. Written in matrix form:

$$\begin{aligned} \mathbf{y} &= \mathbf{Qx} + \mathbf{Wy} \\ \mathbf{y} &= (\mathbf{1} - \mathbf{W})^{-\mathbf{1}}\mathbf{Qx} \end{aligned} \tag{5}$$

During training, the rules for modification of these weights are as follows:

$$\begin{aligned} \Delta w_{ij} &= -\alpha y_i y_j \\ \Delta q_{ij} &= \beta(x_j y_i - q_{ij} y_i^2) \end{aligned} \tag{6}$$

where α and β are the learning rates for each connection.

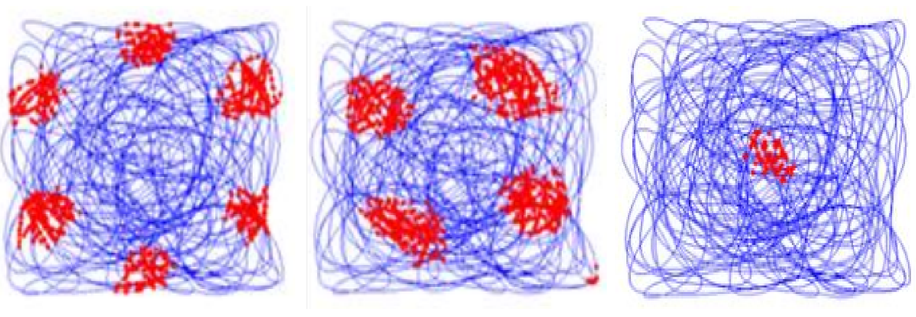
In the full model of the neural architecture, there are two sets of LAHNs - the spatial cell layer LAHN_{SC} and the place cell layer LAHN_{PC}, each producing different types of firing fields. The simplified model, on the other hand, has just one layer of LAHN neurons.

1.1.4 Spatial Cell Patterns

The LAHN neurons are classified based into the following categories based on their firing field patterns:

- **Place cells** - These cells only fire when the animal is in the vicinity of a particular point in the environment.
- **Grid cells** - These cells only fire when the animal is on the vertices of a repeating pattern on the environment. These repeating patterns may be overlapping hexagons or squares.
- **Border cells** - These cells only fire when the animal is in the vicinity of a border of the environment.
- **Plane cells** - These cells only fire when the animal is on a particular 2D plane.

The last two categories are not considered for analysis in this project.



(a) Hexagonal grid cell (b) Square grid cell (c) Place cell

Figure 3: Sample firing fields of different LAHN neurons. Red dots indicate neuron firing at that location

2 Previous Work

Previous experiments and model simulations have been done on two scenarios - one in which the animal is constrained to a 2D plane (rats) and one in which it isn't (Egyptian fruit bats). The same neural architecture is present in both creatures but the firing patterns of the LAHN neurons differ. Techniques to analyze the firing patterns and determine the type of spatial cell involve three main steps - generating neuron firing map, autocorrelation, calculating spatial information index and gridness scores.

2.1 Analysis Techniques

Neuron Firing Map - The neuron firing fields obtained from the simulations contain a set of points where the neuron fires. This needs to be converted into a 2D/3D map of strength of activation of the neurons at these points. The entire environment is divided into bins and if a point in the firing field exists in that bin, its value is increased.

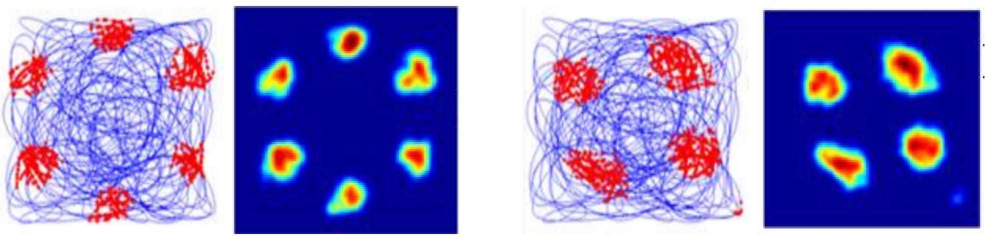


Figure 4: Firing fields and their respective firing maps

Autocorrelation Map - In order to bring out the symmetry present in the firing fields, their autocorrelation is taken according to the following formula:

$$\mathrm{r}(\tau_\mathrm{x},\tau_\mathrm{y}) = \frac{\mathrm{M}\sum_{\mathrm{x,y}}\lambda(\mathrm{x,y})\lambda(\mathrm{x}-\tau_\mathrm{x},\mathrm{y}-\tau_\mathrm{y}) - \sum_{\mathrm{x,y}}\lambda(\mathrm{x,y})\sum_{\mathrm{x,y}}\lambda(\mathrm{x}-\tau_\mathrm{x},\mathrm{y}-\tau_\mathrm{y})}{\sqrt{\left[\mathrm{M}\sum_{\mathrm{x,y}}\lambda(\mathrm{x,y})^2 - \left[\mathrm{M}\sum_{\mathrm{x,y}}\lambda(\mathrm{x,y})\right]^2\right]\left[\mathrm{M}\sum_{\mathrm{x,y}}\lambda(\mathrm{x}-\tau_\mathrm{x},\mathrm{y}-\tau_\mathrm{y})^2 - \left[\sum_{\mathrm{x,y}}\lambda(\mathrm{x}-\tau_\mathrm{x},\mathrm{y}-\tau_\mathrm{y})\right]^2\right]}}$$

where M is the total number of pixels in the firing map, $\lambda(x, y)$ is the value at that location in the firing map, τ_x, τ_y are the coordinate spatial lags between the signal and its copy. This formula for autocorrelation normalizes the values such that the central peak does not dominate. This formula can be extended in a similar manner to 3D space.

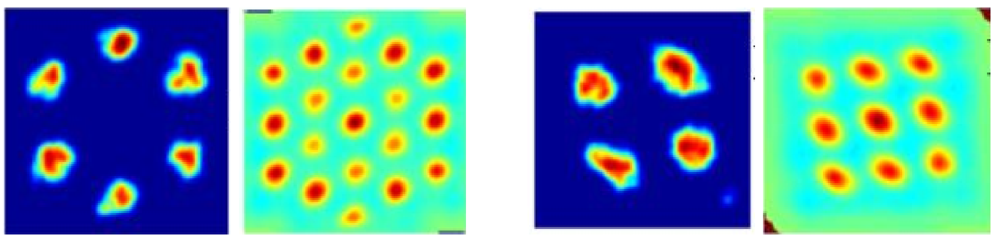


Figure 5: Firing maps and their autocorrelation maps

Spatial Information Index - This is a score that's used to judge if a neuron carries any relevant spatial information. A high SII indicates the neuron fires in very few regions and when combined with other neuron outputs, the position of the animal can be estimated. It is computed as follows:

$$\mathrm{SI} = \sum_{\mathrm{i}} \mathrm{p_i} \frac{\lambda_\mathrm{i}}{\lambda} \log_2 \left(\frac{\lambda_\mathrm{i}}{\lambda} \right) \tag{7}$$

where p_i is the probability of animal being in the i^{th} pixel of the firing map, calculated as number of times the animal visited that pixel to the total time spent in the environment. λ_i is the firing rate of the i^{th} pixel and λ is the mean firing rate over the entire map. For the simulations, if SII>1 the cell was considered a spatial cell.

Grid scores - Once the autocorrelation map is calculated, we can check for hexagonal and square symmetries. To do so, we calculate the Pearson coefficient between rotated versions of the maps and assign hexagon and square grid scores (HGS and SGS) as follows:

$$\begin{aligned} \mathrm{HGS} &= \min\left[\mathrm{cor}(\mathrm{r}, \mathrm{r}^{60^\circ}), \mathrm{cor}(\mathrm{r}, \mathrm{r}^{120^\circ})\right] - \max\left[\mathrm{cor}(\mathrm{r}, \mathrm{r}^{30^\circ}), \mathrm{cor}(\mathrm{r}, \mathrm{r}^{90^\circ}), \mathrm{cor}(\mathrm{r}, \mathrm{r}^{150^\circ})\right] \\ \mathrm{SGS} &= \mathrm{cor}(\mathrm{r}, \mathrm{r}^{90^\circ}) - \max\left[\mathrm{cor}(\mathrm{r}, \mathrm{r}^{45^\circ}), \mathrm{cor}(\mathrm{r}, \mathrm{r}^{135^\circ})\right] \end{aligned} \tag{8}$$

where cor(.) indicates the Pearson cross-correlation coefficient, r - 2D autocorr map, r^θ - autocorr map rotated by θ degrees. In order to be classified as a map with hexagon firing symmetry, HGS should be greater than 0 and SGS less than 0. For square symmetry, its vice-versa. If the cell's spatial information index $>$ 1 but no definite grid patterns can be seen, it is probably another type of cell such as place, border or plane cell.

2.2 Simulations in 2D environment

A virtual animal was allowed to freely navigate a square, 2D environment without any obstacles. Three of the walls were blackened and the fourth contained a white cue card. This acted as the visual input for the rat. For this simulation, the full network model was used. Once the LAHN weights converged, the firing patterns of the LAHN cells were of four types - non-spatial cells, place, grid, border cells. The percentages found in the $\mathrm{LAHN_{SC}}$ layer were as follows:
Hexagon grid cells $\approx$ 37%
Square grid cells $\approx$ 39.6%
Border cells and non-grid cells $\approx$ 24%
This matched the experimental observations performed on the foraging rat's Hippocampus and Endorhinal Cortex.

2.3 Simulations in 3D environment

A virtual animal was once again allowed to freely navigate a cubical enclosure and after the LAHN converged, some of the LAHN neurons evolved distinct 3D firing fields. Analysis of 3D grids is not as straightforward as the 2D case and requires an extra step.

As with the 2D environment, a firing map is populated from the firing fields and its 3D autocorrelation is taken. But calculation of grid scores is only defined for 2D autocorrelation maps. Depending on the 3D structure under scrutiny, different ways of taking 2D slices out of this 3D autocorrelation map are used.

The simulations revealed that 95% of the LAHN neurons evolve as spatial cells. Out of these:
Place cells - 32.43%
Grid cells - 23.97%

Border cells - 28.1%
Plane cells - 15.5%
Unfortunately, there isn't enough experimental data to verify these observations. The fruit bat was engaged in full flight only during recording of place cells. For the grid and border cell recording, the bat was constrained to move only on the floor.

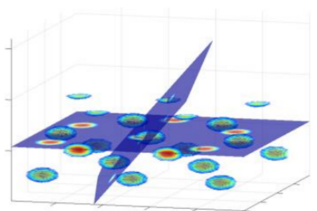
2.3.1 3D grid cell patterns

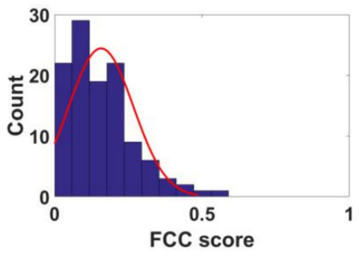
While visualizing 3D grid cell patterns, we usually look at the structure as comprised of spheres stacked in certain patterns (due to its origins in crystallography). Here, we treat local maxima of the autocorrelation map as the centers of these spheres and look for patterns in the arrangement of these spheres.

Face-centered Cubic lattice - The FCC lattice is made of layers of hexagon symmetry stacked on top of each other in an ABCABC fashion i.e. every third layer is exactly identical in terms of orientation of hexagons. In order to determine presence of FCC structure, the following analysis is done:

1. The nearest peak from the center of the 3D autocorrelation map is found. It is assumed that these two peaks are part of the pattern being searched for.
2. A 2D plane is rotated about the line joining these two peaks in steps of 1° and for each of these planes, a grid score is calculated.
3. The plane with the maximum grid score is taken and grid scores for planes at angles of 72° from this one are checked. The average of the top three grid scores is taken.
4. Step 3 is repeated for an analytically generated FCC lattice and the ratio of the two averages is taken to be the "FCC score" for this autocorrelation map.

Upon calculating FCC score for all neurons and fitting a Gaussian distribution on FCC scores, it was found that this simulation *did not produce significant FCC structures.*


(a) Transecting planes at 72° to each other

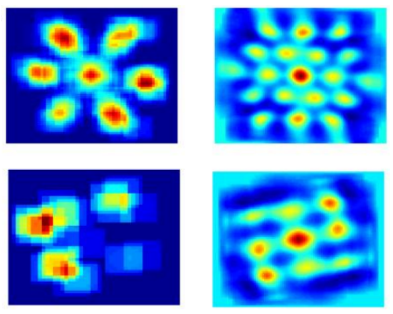


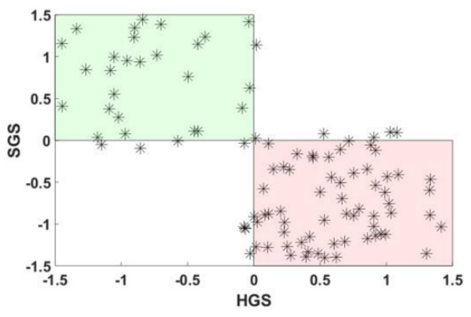
(b) Fitting a Gaussian to distribution of FCC scores

Figure 6: Analysis of FCC symmetry

Planar Symmetry - This structure consists of identical 2D hexagonal or square patterns stacked on top of one another. In order to determine planar symmetry, the projections of the 3D map were taken on the XY, YZ and ZX planes and grid scores were calculated for these 2D maps.
When the grid scores of all the projections were plotted on a scatter plot, a *strong planar symmetry was confirmed* for the neurons in this model. Among the neurons whose grid scores were calculated:
Hexagon symmetry - 60%

Square symmetry - 26%
Undetermined - 14%
In the figure given below, the green area indicates square symmetry and the red indicates hexagon symmetry.


(a) Sample firing map and autocorrelation of projections onto 2D coordinate planes



(b) Scatter plot of grid scores of projections

Figure 7: Analysis of Planar Symmetry

2.3.2 Assumptions and Simulation Parameters

- Owing to the absence of sharp ascents or descents in the trajectory of a bat, the variance is pitch is much smaller than the variance in the azimuth(a uniform distribution). Hence the trajectory generated during simulations had a similar distribution. This is also the reason why a 70:30 ratio was observed for the number of azimuthal:pitch head direction cells.
- A total of 50 LAHN neurons and 100 HD neurons were used.
- Thresholds for classifying cell types:
  Spatial cell: SII > 1
  Hexagonal Grid cell: HGS>0 and SGS<0
  Square Grid cell: HGS<0 and SGS>0

3 Addition of Obstacles in 3D space

This project is an attempt to simulate this model in a 3D environment with obstacles such as a library or a warehouse. The virtual animal is replaced by a virtual drone such as a quadcopter which can freely move along all directions. Just like the empty 3D environment under consideration previously, there exists no experimental data to verify the outcomes of this model. These simulations only predict the possible patterns of spatial cells under these conditions. All simulations and analysis is done using MATLAB 2016a.
All obstacles under consideration are simple rectangular obstacles which can be modelled in MATLAB with a few simple parameters.

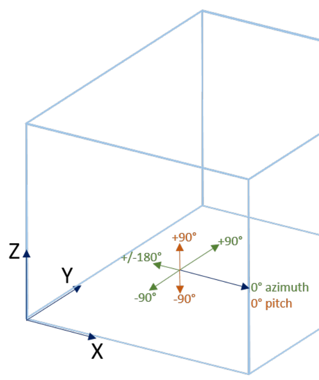
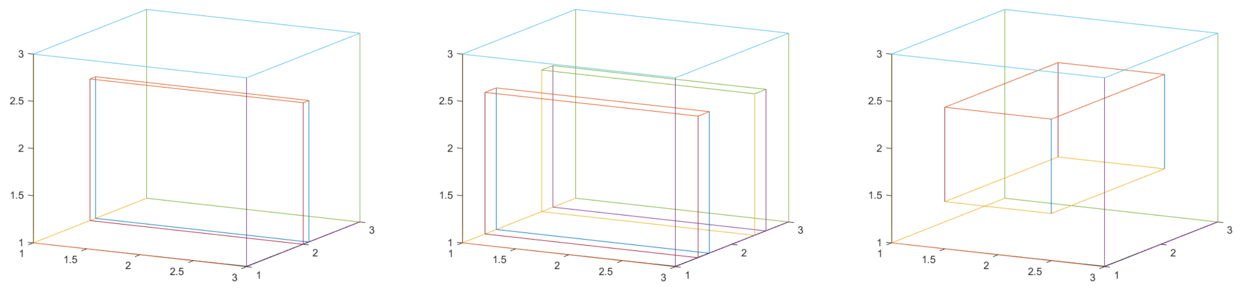


Figure 8: The 3D coordinte system used in simulations

Unsymmetric obstacles
Two shelf-like obstacle sets have been considered. One is a single shelf that runs along the length of the Y axis and another consists of two such shelves. They are unsymmetric in the sense that the dimensions of obstacles and position along the three axes are not the same.
Symmetric obstacles
In order to maintain symmetry along the X and Z axes, a floating cuboidal obstacle has been chosen for the simulations.



(a) Single shelf (b) Two shelves (c) Floating cuboid

Figure 9: 3D views of obstacles

4 Random Path Generation

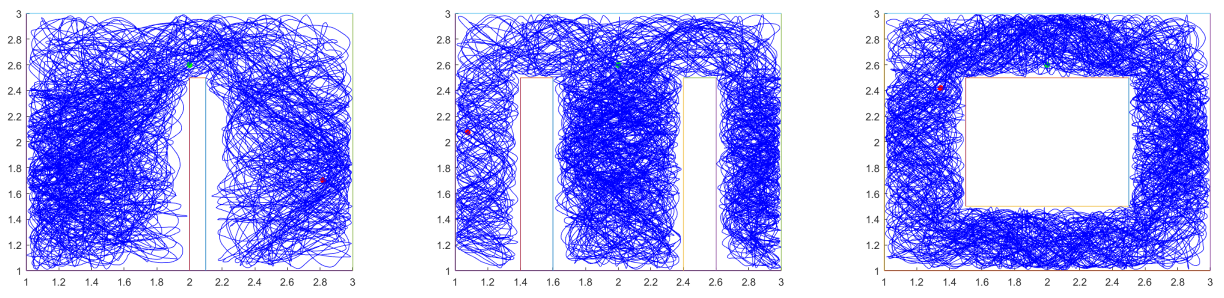
Since this simulation is carried out for an artificial drone, pitch and azimuth can be varied freely. Hence the process is greatly simplified.

1. Define the walls of the enclosure and the obstacles in MATLAB using simple 3D coordinates for each surface of the obstacle

2. At any given time instant, a random acceleration value is chosen from a normal distribution centered about zero. This is then integrated to get velocity and position in the next time step using simple Euler integration.

3. In order to prevent collision with obstacles and maintain a smooth path at the same time, the distance from each obstacle is calculated at every time instant and an acceleration inversely proportional to this distance is added to the randomly generated acceleration values.

4. To reduce the chances of the virtual drone crossing an obstacle, the time resolution is chosen as a small value (say 0.01s).



(a) Single shelf path (b) Two shelves path (c) Floating cuboid path

Figure 10: Random paths around obstacles

Listing 1: Defining obstacles and boundaries

```
% each obstacle is a rectangle with 4 points defining it (total 12 params)
% params order - xyz1, xyz2, xyz3, xyz4
shelves = [];
% defining boundaries in this form
max_dim = [3, 3, 3];
min_dim = [1, 1, 1];
x = min_dim(1); X = max_dim(1);
y = min_dim(2); Y = max_dim(2);
z = min_dim(3); Z = max_dim(3);
shelves = [shelves; [x,y,z, X,y,z, X,Y,z, x,Y,z]]; %bottom
shelves = [shelves; [x,y,z, X,y,z, X,y,Z, x,x,Z]]; %face1
shelves = [shelves; [x,y,z, x,y,Z, x,Y,Z, x,Y,z]]; %face2
shelves = [shelves; [X,y,z, X,y,Z, X,Y,Z, X,Y,z]]; %face3
shelves = [shelves; [x,Y,z, x,Y,Z, X,Y,Z, X,Y,z]]; %face4
shelves = [shelves; [x,y,Z, X,y,Z, X,Y,Z, x,Y,Z]]; %top
% obstacle1 - floating cuboid
shelves = [shelves; [1.5,1,1.5, 1.5,1,2.5, 1.5,3,2.5, 1.5,3,1.5]]; % face1
shelves = [shelves; [2.5,1,1.5, 2.5,1,2.5, 2.5,3,2.5, 2.5,3,1.5]]; % face2
shelves = [shelves; [1.5,1,2.5, 1.5,3,2.5, 2.5,3,2.5, 2.5,1,2.5]]; % top
shelves = [shelves; [1.5,1,1.5, 1.5,3,1.5, 2.5,3,1.5, 2.5,1,1.5]]; % bottom
%
% similarly, other obstacles are defined
%
```

Listing 2: Checking for obstacles and updating acceleration

```
function updated = check_obs(ppos, pvel, npos, nvel, shelves, unit_vecs, ↩
    min_dim, max_dim)

pos = npos; vel = nvel;
% limit velocity
```

```
vel(vel>0.2) = 0.2;
vel(vel<-0.2) = -0.2;
% check for box limits
inds = find(npos>=max_dim);
pos(inds) = ppos(inds);
vel(inds) = -pvel(inds);
inds = find(npos<=min_dim);
pos(inds) = ppos(inds);
vel(inds) = -pvel(inds);

% check if crossed over for each obstacle and update them
acc_up = [0,0,0];
deflection = [0.01 0.01 0.01];
for i = 1:size(shelves,1)
    % check which dimension is to be checked
    dim = find(unit_vecs(i,:)~=0);
    xcheck = npos(1)>max(shelves(i,[1,4,7,10])) || npos(1)<min(shelves(i↩
        ,[1,4,7,10]));
    ycheck = npos(2)>max(shelves(i,[2,5,8,11])) || npos(2)<min(shelves(i↩
        ,[2,5,8,11]));
    zcheck = npos(3)>max(shelves(i,[3,6,9,12])) || npos(3)<min(shelves(i↩
        ,[3,6,9,12]));

    if((dim==3)&&(xcheck||ycheck)) % check x and y limits
        continue;
    elseif((dim==1)&&(ycheck||zcheck)) % check y and z limits
        continue;
    elseif((dim==2)&&(xcheck||zcheck)) % check x and z limits
        continue;
    else
        % add up deflection and reflect if it reaches edge
        if(npos(dim)>=shelves(i,dim))
            if(ppos(dim)<shelves(i,dim))
                pos(dim) = ppos(dim);
                vel(dim) = -vel(dim);
            else
                acc_up(dim) = acc_up(dim) + deflection(dim)/(npos(dim)-shelves↩
                    (i,dim));
            end
        elseif(npos(dim)<=shelves(i,dim))
            if(ppos(dim)>shelves(i,dim))
                pos(dim) = ppos(dim);
                vel(dim) = -vel(dim);
            else
                acc_up(dim) = acc_up(dim) + deflection(dim)/(npos(dim)-shelves↩
                    (i,dim));
            end
        end
    end
end
updated = [pos;vel;acc_up];
```

```
end
```

5 Simulation to train the network

The input to the simulation program is an array of 3D points generated in the previous step. The program aims to train the LAHN network and output the set of LAHN weights as well as the firing fields of each neuron. There are 3 main steps involved:

5.1 Preparing the input for LAHN

The position data is first converted into velocity(magnitude) and direction data. The direction data will be fed to the HD layer and the velocity will be fed to the PI layer.

Listing 3: Calculating speed and direction

```
% pos - set of 3D positions given as input
% calculating azimuth and pitch angles at each time instant
delx = diff(pos(:,1)); delx(end+1)=0;
dely = diff(pos(:,2)); dely(end+1)=0;
delz = diff(pos(:,3)); delz(end+1)=0;
theta_az = rad2deg(atan2(dely,delx));headdir_az=deg2rad(theta_az);
theta_pitch = rad2deg(atan2(delz,sqrt(delx.^2+dely.^2)));headdir_pitch=deg2rad↩
    (theta_pitch);
% calculating speed at each time instant
speed=zeros([1,size(pos,1)-1]);
for ii=2:size(pos,1)
    speed(ii-1) = pdist2(pos(ii-1,:),pos(ii,:));
end
s_az=speed;s_pitch=speed;
```

Next, the outputs of HD cells and PI cells are calculated and populated for training the LAHN. The ratio of HD azimuthal:pitch cells is set to 1:1 and the total number of neurons is seto to 100. The threshold value for the PI layer neurons is set to 0.75.

Listing 4: Calculating LAHN inputs

```
% calculating number of cells
n=100; % total number of hd cells
n_Az_fraction = 50/100; n_pitch_fraction = 1-n_Az_fraction; % ratio of Azimuth↩
     and Pitch neurons.
n_Az = n_Az_fraction*n;
n_pitch = floor(n_pitch_fraction*n)-1;
% defining preferred direction of HD cells
dth_Az=360/n_Az;
dth_pitch=360/n_pitch;
theta_pref_deg_Az=0:dth_Az:360-dth_Az;
theta_pref_Az=deg2rad(theta_pref_deg_Az);
```

```
theta_pref_deg_pitch=0:dth_pitch:360-dth_pitch;
theta_pref_pitch=deg2rad(theta_pref_deg_pitch);

%% PI Azimuthal layer
basefreq=1;
speed_az=s_az';
dendphase=0;
dt=0.1;
betaa=2; %Frequency modulation factor
dendfreq=(basefreq*(ones(length(pos)-1,n_Az)))+betaa*repmat(speed_az,1,n_Az).*↩
    cos(repmat(theta_pref_Az,length(pos)-1,1)-repmat(headdir_az(1:end-1),1,↩
    n_Az));

X = zeros(n_Az,1); Y = ones(n_Az,1);
piosc_az=[];
for ii=1:length(pos)-1
    dendphase=dendphase+2*pi*dendfreq(ii,:)*dt;
    dendosc=sin(dendphase);
    piosc_az(:,ii)=dendosc';
end

%% PI Pitch layer
speed_pitch=s_pitch';
dendphase=0;
RBP = 100;
betaa2=RBP*betaa/100;
dendfreq=(basefreq*(ones(length(pos)-1,n_pitch)))+betaa2*repmat(speed_pitch,1,↩
    n_pitch).*cos(repmat(theta_pref_pitch,length(pos)-1,1)-repmat(↩
    headdir_pitch(1:end-1),1,n_pitch));

Xp = zeros(n_pitch,1); Yp = ones(n_pitch,1);
piosc_pitch=[];
for ii=1:length(pos)-1
    dendphase=dendphase+2*pi*dendfreq(ii,:)*dt;
    dendosc=sin(dendphase);
    piosc_pitch(:,ii)=dendosc';
end

%% Stacking piosc_Az and piosc_pitch and thresholding
piosc_tot = [piosc_az ;piosc_pitch];
piosc_thresh_tot=((piosc_tot)>0.75).*piosc_tot;
```

5.2 Training the LAHN

To improve the training, the thresholded PI layer values are mean subtracted before start of training. All training operations are done in matrix form to speed up calculations. LAHN training equations written in matrix form are:

$$
\begin{aligned}
\mathrm{T}^{(\mathrm{t})} &= \left(1 - \mathrm{W}^{(\mathrm{t})}\right)^{-1} \mathrm{Q}^{(\mathrm{t})} \\
\mathrm{C}_{\mathrm{Y}}^{(\mathrm{t})} &= \mathrm{T}^{(\mathrm{t})} \mathrm{C}_{\mathrm{X}} \mathrm{T}^{(\mathrm{t})\mathrm{T}} \\
\mathrm{W}^{(\mathrm{t}+1)} &= \mathrm{W}^{(\mathrm{t})} - \alpha.\mathrm{offdiag}\left(\mathrm{C}_{\mathrm{Y}}^{(\mathrm{t})}\right) \\
\mathrm{Q}^{(\mathrm{t}+1)} &= \mathrm{Q}^{(\mathrm{t})} + \beta\left(\mathrm{T}^{(\mathrm{t})}\mathrm{C}_{\mathrm{X}} - \mathrm{diag}\left(\mathrm{C}_{\mathrm{Y}}^{(\mathrm{t})}\right)\mathrm{Q}^{(\mathrm{t})}\right)
\end{aligned}
\tag{9}
$$

where $\mathrm{var}^{(\mathrm{t})}$ is the variable at time t. A total of 50 LAHN neurons are present. The threshold for LAHN convergence is set to 0.001x(learning rate) for both kinds of weights. A limit of 2 million is set on the number of iterations.

Listing 5: LAHN training function

```
function [T,Q,W, InfoTransferRatio] = foldiak_linear_fn(X, alphaa, betaa, ↩
    output_neuron_nmbr, maxiter)

X=removemean(X);
[N K] = size(X); %N --> Dimension    K---> # of samples

%% initialize weights
Q=rand(N,output_neuron_nmbr)-0.5;   % Feedforward weights
W=zeros(output_neuron_nmbr,output_neuron_nmbr); %Lateral weights
Y=zeros(output_neuron_nmbr,K);

%% Main loop
ii = 1;
convergeflag = 1;
while(1)
    T = inv(eye(output_neuron_nmbr)-W)*Q';
    CY = T * CX * T';
    offdiagCY = CY;  offdiagCY(logical(eye(size(offdiagCY)))) = 0;
    diagCY = CY.*eye(output_neuron_nmbr);
    dW = offdiagCY;
    dQ = (T*CX - diagCY*Q');
    W = W - alphaa*dW;
    Q = Q + (betaa*dQ)';

    if(mod(ii,10000)==0)
        disp(ii);
    end

    ii = ii + 1;
    dW_max = max(max(abs(dW))); dQ_max = max(max(abs(dQ)));
    if(dW_max<0.001 && dQ_max<0.001)
        convergeflag = 1;
        break;
    end
    if ii > maxiter
        convergeflag = 0;
        break;
    end
```

```
end
```

5.3 Populating the firing fields

The position vectors are simply multiplied by the transformation matrix T to get the values of each neuron. This is then thresholded at 75% of the maximum value to get the firing fields of each neuron. The firing fields are acquired in the form of 3D points as well as projections on XY, YZ, ZX planes.

Listing 6: Populating firing fields

```
%% Get LAHN outputs for each neuron
neuron_number=1:50;
ot1_mat=[]; firposgrid=[];
for ii=1:length(neuron_number)
    w=T(ii,:); %Selecting the weights of that neuron
    ot=w*piosc_thresh_tot; ot=ot'; ot1_mat(:,ii) = ot;
    thresh=max(ot)*.75;
    firr=find((ot)>thresh);
    pos1 = pos_az;
    pos2 = [pos(:,1) pos(:,3)];
    pos3 = [pos(:,2) pos(:,3)];
    firposgrid1=pos1(firr,:);
    firposgrid2=pos2(firr,:);
    firposgrid3=pos3(firr,:);
    firposgrid=pos(firr,:);
    %%%
    % further analysis is done inside this for loop itself
    %%%
end
```

6 Analysis of Firing Fields

The analysis of populated firing fields involves similar techniques to previous experiments - generating firing maps, autocorrelation maps and calculating grid scores.

6.1 Generating Firing rate maps

A firing rate map is a representation of how many times a neuron has fired in a given volume. This function accepts the firing fields as input and creates a 3D firing rate map. An nX x nY x nZ 3D rate map of zeros is first created. X,Y,Z are the dimensions of the environment and n is the resolution into which the space is divided. For each coordinate in the input firing field, the corresponding bin (oe voxel) in the rate map is incremented.

Listing 7: Generating firing rate maps

```
function map = firing_map(field, resolution, limits1, limits2, limits3)

scale = 1/resolution;
map = zeros((limits1(2)-limits1(1))*scale, (limits2(2)-limits2(1))*scale, (↩
    limits3(2)-limits3(1))*scale);

bins = floor(field*scale);
bins(:,1) = bins(:,1) - limits1(1)*scale + 1;
bins(:,2) = bins(:,2) - limits2(1)*scale + 1;
bins(:,3) = bins(:,3) - limits3(1)*scale + 1;

for i = 1:size(field,1)
    map(bins(i,1),bins(i,2),bins(i,3)) = map(bins(i,1),bins(i,2),bins(i,3)) + ↩
        1;
end
end
```

6.2 Generating 3D autocorrelation map

The previously mentioned formula for autocorrelation is used here to calculate the map. The edges of the map with small overlap are ignored and set to zero.

Listing 8: Generating autocorrelation map

```
% primary function
function Rxy = correlation_map_3d(map1,map2)
bins = size(map1,1);
N = bins + round(0.64*bins);
if ~mod(N,2)
    N = N - 1;
end
% Centre bin
cb = (N+1)/2;
Rxy = zeros(N,N,N);
for ii = 1:N
    rowOff = ii-cb;
    for jj = 1:N
        colOff = jj-cb;
        for kk = 1:N
            chanOff = kk-cb;
            Rxy(ii,jj,kk) = pointCorr3d(map1,map2,rowOff,colOff,chanOff,bins);
        end
    end
end
end

% helper function pointCorr
function Rxy = pointCorr3d(map1,map2,rowOff,colOff,chanOff,N)
```

```
% Number of rows in the correlation for this lag
numRows = N - abs(rowOff);
% Number of columns in the correlation for this lag
numCol = N - abs(colOff);
% Number of channels in the correlation for this lag
numCh = N - abs(chanOff);

% Set the start and the stop indexes for the maps
if rowOff > 0
    rSt1 = 1+abs(rowOff)-1;
    rSt2 = 0;
else
    rSt1 = 0;
    rSt2 = abs(rowOff);
end
if colOff > 0
    cSt1 = abs(colOff);
    cSt2 = 0;
else
    cSt1 = 0;
    cSt2 = abs(colOff);
end
if chanOff > 0
    chSt1 = abs(chanOff);
    chSt2 = 0;
else
    chSt1 = 0;
    chSt2 = abs(chanOff);
end

sumXY = 0;
sumX = 0;
sumY = 0;
sumX2 = 0;
sumY2 = 0;
NB = 0;
for ii = 1:numRows
    for jj = 1:numCol
        for kk = 1:numCh
            if ~isnan(map1(rSt1+ii,cSt1+jj,chSt1+kk)) && ~isnan(map2(rSt2+ii,↩
                cSt2+jj,chSt2+kk))
                NB = NB + 1;
                sumX = sumX + map1(rSt1+ii,cSt1+jj,chSt1+kk);
                sumY = sumY + map2(rSt2+ii,cSt2+jj,chSt2+kk);
                sumXY = sumXY + map1(rSt1+ii,cSt1+jj,chSt1+kk) * map2(rSt2+ii,↩
                    cSt2+jj,chSt2+kk);
                sumX2 = sumX2 + map1(rSt1+ii,cSt1+jj,chSt1+kk)^2;
                sumY2 = sumY2 + map2(rSt2+ii,cSt2+jj,chSt2+kk)^2;
            end
        end
    end
```

```
end

if NB >= 4*4*4
    sumx2 = sumX2 - sumX^2/NB;
    sumy2 = sumY2 - sumY^2/NB;
    sumxy = sumXY - sumX*sumY/NB;
    if (sumx2<=0 && sumy2>=0) || (sumx2>=0 && sumy2<=0)
        Rxy = 0;
    else
        Rxy = sumxy/sqrt(sumx2*sumy2);
    end
else
    Rxy = 0;
end
```

6.3 Taking 2D slices from 3D autocorrelation map

Three rotation axes passing through the center peak and parallel to each of the axes are considered for taking 2D slices. The autocorr map is rotated in steps of 2 degrees and the central plane is taken as the rotated 2D slice. For each angle, the autocorrelation map is rotated about the three axes and three slices are taken from its center. Therefore, for each 3D map, we get (180/2)x3 = 270 2D slices for which we calculate grid scores. The 3D map is treated as an image to exploit the image processing functions in MATLAB.

Listing 9: 2D slices

```
% primary function
function [map2dx, map2dy, map2dz] = rotated_slices(map3d, angle)

size3d = size(map3d);
if(length(size3d)==2)
    size3d = [size3d 1];
end
% add 1 to every value - needed to trim off zeros after rotation
map3d = map3d + 1;

%% rotate map3d about Z axis (Z coord remains same, XY transformed)
zrot = imrotate(map3d,-angle);
map2dz = extract_middle_row(zrot);
% subtract the 1 initially added
map2dz = map2dz - 1;

%% rotate map3d about Y axis (Y coord remains same, ZX transformed)
% switch axes
map3d_xzy = zeros(size3d(2), size3d(3), size3d(1)); % X -> rows, Z -> cols, Y ↩
    -> depth
for ii = 1:size3d(1)
    map3d_xzy(:,:,ii) = map3d(ii,:,:);
end
```

```
yrot = imrotate(map3d_xzy, -angle);
map2dy = extract_middle_row(yrot);
% subtract the 1 initially added
map2dy = map2dy - 1;

%% rotate map3d about X axis (X coord remains same, YZ transformed)
% switch axes
map3d_yzx = zeros(size3d(1),size3d(3),size3d(2)); % Y -> rows, Z -> cols, X ->↩
    depth
for ii = 1:size3d(2)
    map3d_yzx(:,:,ii) = map3d(:,ii,end:-1:1); % Z reversed
end
xrot = imrotate(map3d_yzx,-angle);
map2dx = extract_middle_row(xrot);
% subtract the 1 initially added
map2dx = map2dx - 1;
end

% helper function - extract_middle_row
function map2d = extract_middle_row(map3d)

size3d = size(map3d);
if(mod(size3d(1),2)) % odd number of rows - just select middle row values
    map2d = map3d((size3d(1)+1)/2, :, :);
else % take average of the middle two rows
    map2d = (map3d(size3d(1)/2, :, :) + map3d(size3d(1)/2+1, :, :))/2;
end
size2d = size(map2d);
if(length(size2d)==2)
    size2d = [size2d 1];
end
map2d = reshape(map2d, [size2d(2),size2d(3)]);
% trim zeros
first_ind = find(map2d(:,1),1,'first');
last_ind = find(map2d(:,1),1,'last');
map2d = map2d(first_ind:last_ind, :);
end
```

The diagram below explains the idea behind taking the 2D slices:

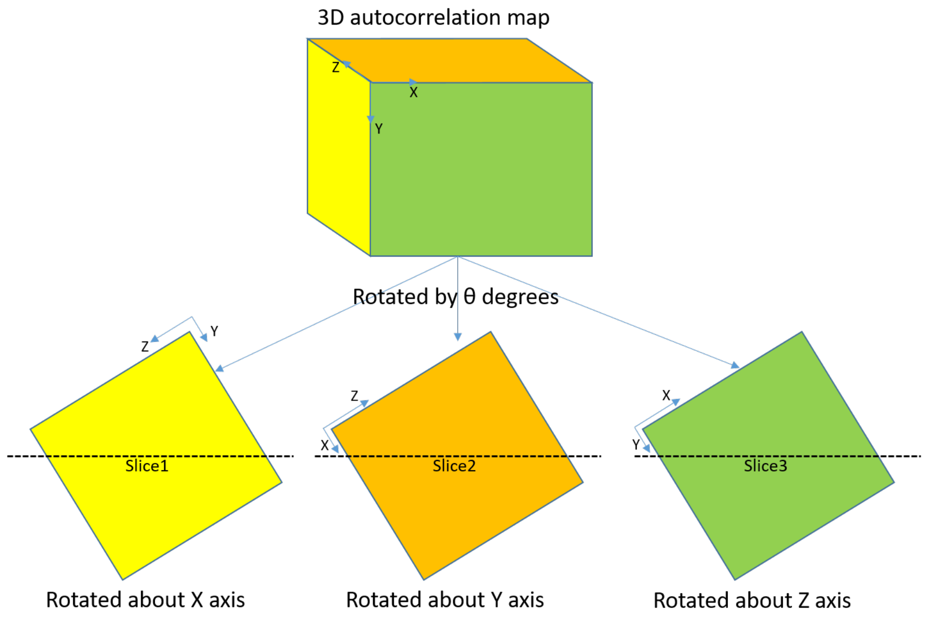


Figure 11: Taking three 2D slices for each angle

6.4 Calculating Grid Scores

As explained before, each map gets a hexagon and square grid score and depending on these, it gets classified. The rotation of the 2D maps is about its center and the smallest common area is taken for calculation of Pearson coefficient. The maps are interpolated as necessary in order to get a properly centered overlap between the rotated maps.

Listing 10: Grid Scores

```
% primary function
function [hgs, sgs] = grid_scores(map)

%% calculate square grid score
map90 = imRotateCrop(map, 90);
map45 = imRotateCrop(map, 45);
map135 = imRotateCrop(map, 135);
dim_sq = min([size(map90);size(map45);size(map135)]);
trim_orig = trim_matrix(map, dim_sq);
cor1 = corrcoef(trim_orig,trim_matrix(map90,dim_sq));
cor2 = corrcoef(trim_orig,trim_matrix(map45,dim_sq));
cor3 = corrcoef(trim_orig,trim_matrix(map135,dim_sq));
sgs = cor1(2) - max([cor2(2),cor3(2)]); % cor(1) - self coeff. cor(2) - cross ↩
    coeff.

%% calculate hexagon grid score
map60 = imRotateCrop(map, 60);
map120 = imRotateCrop(map, 120);
map30 = imRotateCrop(map, 30);
map150 = imRotateCrop(map, 150);
dim_hex = min([size(map90);size(map60);size(map120);size(map30);size(map150)])↩
```

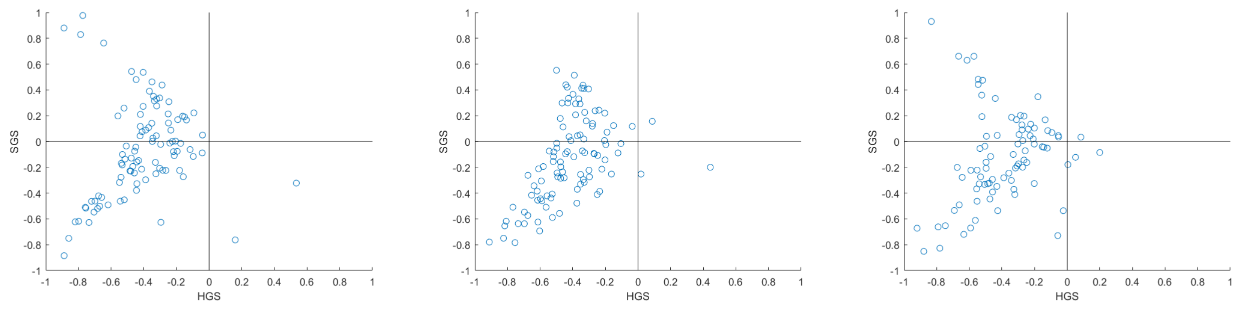
```
    ;
trim_orig = trim_matrix(map, dim_hex);
cor1 = corrcoef(trim_orig,trim_matrix(map90,dim_hex));
cor2 = corrcoef(trim_orig,trim_matrix(map60,dim_hex));
cor3 = corrcoef(trim_orig,trim_matrix(map120,dim_hex));
cor4 = corrcoef(trim_orig,trim_matrix(map30,dim_hex));
cor5 = corrcoef(trim_orig,trim_matrix(map150,dim_hex));
hgs = min([cor2(2),cor3(2)]) - max([cor4(2),cor1(2),cor5(2)]);

end

% helper function - trim_matrix
function trimmed = trim_matrix(mat, dim)

size_in = size(mat);
%% if sizes are same dont do anything
if(dim(1)==size_in(1) && dim(2)==size_in(2))
    trimmed = mat;
    return;
end

[Xin,Yin] = meshgrid(1:size_in(2), 1:size_in(1));
%% check if interpolation is needed
% if both row dims and column dims are diff type
if((mod(size_in(1),2) && ~mod(dim(1),2) && mod(size_in(2),2) && ~mod(dim(2),2)↩
    ) || ...
        (~mod(size_in(1),2) && mod(dim(1),2) && ~mod(size_in(2),2) && mod(dim↩
            (2),2)))
    [Xq,Yq] = meshgrid(1:0.5:size_in(1), 1:0.5:size_in(2));
    mat_int = interp2(Xin,Yin,mat,Xq,Yq);
    mat = mat_int(2:2:end, 2:2:end); % alternate rows, alternate columns
% if row dimensions are diff type but column dimensions are same type
elseif((mod(size_in(1),2) && ~mod(dim(1),2)) || (~mod(size_in(1),2) && mod(dim↩
    (1),2)))
    [Xq,Yq] = meshgrid(1:size_in(2), 1:0.5:size_in(1));
    mat_int = interp2(Xin,Yin,mat,Xq,Yq);
    mat = mat_int(2:2:end, :); % alternate rows, all columns
% if row dimensions are same type but column dims are diff type
elseif((mod(size_in(2),2) && ~mod(dim(2),2)) || (~mod(size_in(2),2) && mod(dim↩
    (2),2)))
    [Xq,Yq] = meshgrid(1:0.5:size_in(2), 1:size_in(1));
    mat_int = interp2(Xin,Yin,mat,Xq,Yq);
    mat = mat_int(:, 2:2:end); % all rows, alternate columns
end

%% now trim the matrix accordingly
size_in = size(mat);
% if both final size and mat size are odds
if(mod(size_in(1),2) && mod(dim(1),2) && mod(size_in(2),2) && mod(dim(2),2))
    center = (size_in+1)/2;
    radius = (dim-1)/2;
    trimmed = mat(center(1)-radius(1):center(1)+radius(1), center(2)-radius(2)↩
       :center(2)+radius(2));
% if both final size and mat size are evens
else
    center = size_in/2;
    radius = dim/2-1;
    trimmed = mat(center(1)-radius(1):center(1)+1+radius(1), center(2)-radius↩
       (2):center(2)+1+radius(2));
end
end
```

7 Results

For each neuron, its Spatial Information Index is first calculated and only if it is greater than 1, any further analysis is done on it. For finding grid cell patterns, we take,as before, 2D slices from the 3D map as well as projections onto 2D surfaces to evaluate symmetry. It was observed that 90%-95% of all LAHN neurons had SII>1. The predominant type of symmetry found in all three simulations was **planar symmetry** with **square patterns**.

7.1 Projections onto 2D planes

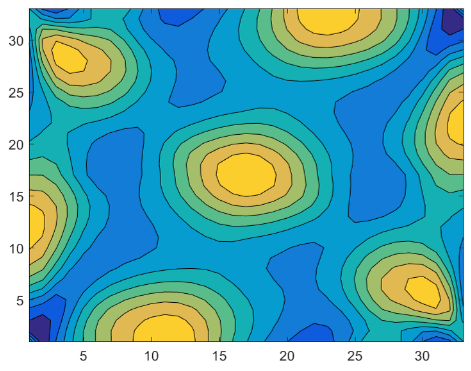
For each neuron, the grid scores for all three projections are calculated and plotted on a scatter plot:



(a) Single Shelf (b) Double Shelf (c) Floating Cuboid

Figure 12: Scatter plots of grid scores (projections)

From these figures, we can see that there is hardly any neuron which exhibits planar symmetry with hexagon patterns. But 30%-40% of all spatially relevant neurons are exhibiting square planar symmetry.


(a) Hexagon pattern

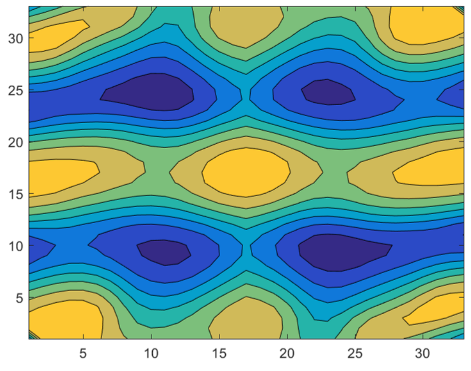
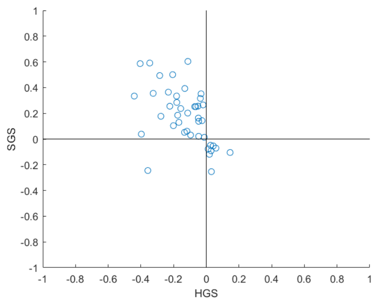
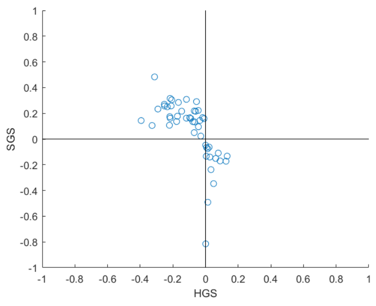

(b) Square pattern

Figure 13: Sample grid patterns neurons when projected on 2D planes

7.2 2D slices

For each neuron, the scores of the planes with the best hexagon score and the best square score are populated and a scatter plot was created with them:


(a) Single Shelf


(b) Double Shelf

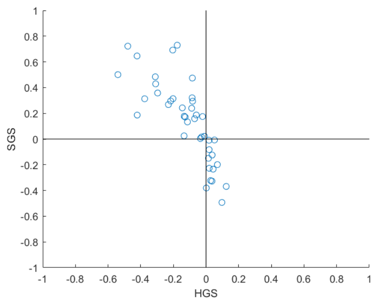
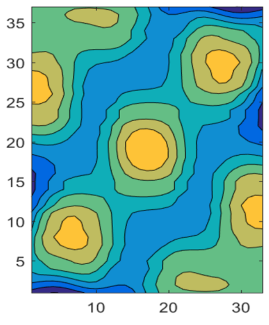

(c) Floating Cuboid

Figure 14: Scatter plots of grid scores (slices)

From these figures, we cannot draw sufficient conclusions to the nature of grid patterns as, once again, we see more of square symmetry than hexagon symmetry, which does not confirm FCC or HCP structures. The grid scores recorded, though above the specified thresholds, are weak compared to previous observations in other works.


(a) Hexagon pattern

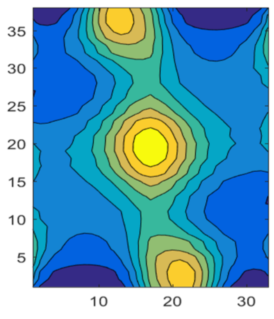

(b) Square pattern

Figure 15: Sample grid patterns neurons when 2D planes are transected from 3D maps

7.3 Interpreting the Results

The key observation is the strong presence of planar square symmetry. This departs from the previous observations of hexagonal planar symmetry in the no obstacles case. There may be several reasons for this observation:

- **Pitch variance** - this model assumes free, uniform pitch variance as opposed to previous work with constrained variance. This may drive the grid cells to assume planar square patterns. When all constraints on movement are removed, such as in drones, a repeating pattern identical in all directions makes the most sense. As opposed to this, animals prioritize location on the azimuth plane over their altitude. This could lead to a more complex system of self-location in the azimuthal direction compared to altitude.
- **Consequence of small environment** - The simulation is carried out in a 2x2x2 box with a resolution of 0.1 for the firing rate map. As a result, peaks showing symmetry may not have appeared or may have been cut out while finding grid scores. A bigger environment with finer resolution could reveal more symmetries.
- **Too few LAHN neurons** - Since the environment, and hence the animal's path, is more complex, the limited number of LAHN neurons may be forced to learn simpler representations. Introduction of more LAHN neurons could give the network the ability to learn more complex representations such as FCC and Hexagonal Close Packing (HCP)(Stella and Treves 2015, Mathis et al 2015).
- **Local vs Global grid scores** - In the above analysis, grid scores were only calculated on the autocorrelation map of the whole firing field. Some previous works have simulated 2D environments with obstacles and concluded that as the size of obstacles increase, the network begins to learn the presence of the separate regions. Some neurons only fire when the animal is in one region, but they may also have hexagonal and square patterns. Similarly, a region wise analysis could be done in this case as well to determine if neurons form grid cell patterns within those volumes.

The ultimate aim of this project was to investigate the grid cell patterns formed in 3D environments with obstacles under unconstrained movement (such as a drone in a library or warehouse). Further simulations with the above changes can help pinpoint the architecture which best suits our purpose. Once the architecture is finalized, one can build specialized systems (using mutable parallel hardware such as FPGAs) for real-time implementation of these networks on vehicles such as drones. These nature-inspired systems will be more accurate and computationally efficient compared to existing neural network based systems.

References

[1] K.Soman, V.Muralidharan, S.Chakravarthy. *AN OSCILLATORY NETWORK MODEL OF HEAD DIRECTION, SPATIALLY PERIODIC CELLS AND PLACE CELLS USING LOCOMOTOR INPUTS.* 2016.

[2] K.Soman, S.Chakravarthy, M.M.Yartsev. *A Hierarchical Anti-Hebbian Network Model for the Formation of Spatial Cells in Three-Dimensional Space.* 2018.

[3] K.Soman, V.Muralidharan, S.Chakravarthy. *A Model of Multisensory Integration and its Influence on Hippocampal Spatial Cell Responses.* IEEE 2017.

[4] K.Soman, V.Muralidharan, S.Chakravarthy. *A unified hierarchical oscillatory network model of head direction cells, spatially periodic cells, and place cells*. European Journal of Neuroscience 2018.

[5] P. Foldiak. *Adaptive Network for optimal Linear Feature Extraction*. 1989.

[6] David Young. *MATLAB helper function imRotateCrop()* [Online].
Available: https://in.mathworks.com/matlabcentral/fileexchange/48624-rotate-images-with-automatic-cropping