**Technical Appendix 1 – Example Arena Experiments and Code**

The following instructions assume a standard arena setup with 44 panels (four rows and eleven columns), MATLAB is installed, and the”controller” folder from the Panels bitbucket website (https://bitbucket.org/mreiser/panels) is included in your MATLAB path. Going from idea to execution, this section lists the main steps and is intended as a standalone example. Note that more information on troubleshooting and setup is available on the Panels bitbucket wiki site listed in the introduction.

Plan a stimulus.

What type of experiment is desired? If there is literature already using the desired stimulus, then methods sections might be a good place to start. Even if there is not a history of your stimulus being used, this might be a good starting point if it is unclear exactly what happens in flight arena experiments. Once you have a good idea, drawing out the pattern is helpful for the next step. For the example we have chosen a simple expansion-contraction pattern, where a grating will expand from some position and contract 180° from it. For the experiment, we will be testing how the position of the expansion or contraction impacts the turning response. Specifically, we will want to test the turning response at one of three foci of expansion or contraction.

Design the pattern(s).

Once you know what stimulus you want to use, you must create it using a MATLAB m-file. Open a new m-file in the MATLAB editor, and name it something related to your stimulus, to keep things organized name it starting with ‘make\_’ followed by a description. In order to generate a pattern from this file, a few parameters must be defined. The Software Overview section has information on the variables being set in this example; here we will reiterate the points made. For our expansion example, if thinking of the X and Y channels as axes of a memory buffer, the X channel will be all the different possible frames of expansion from one point (say, directly in front of the fly). The Y channel can then be all of the different shifts, or starting positions for this expansion. Note that each of the X-Y positions now corresponds to a state, or frame, of the arena. Here is how this is done.

First we need to populate and save a MATLAB structure called pattern and save the pattern to a file for later use.

pattern.x\_num = 96; % x is all frames of one expansion

pattern.y\_num = 96; % y is different starting positions of expansion, one for each position in the arena

pattern.num\_panels = 48;

pattern.gs\_val = 3; % So we can play around with brightness later

pattern.row\_compression = 1; % this can be used here because all of the columns in our pattern can be easily represented by a single row of the arena LEDs

Now we have set the stage for our pattern. There are 96 positions in each axis of our memory buffer. Because the pattern could also be represented by one row of the panels, we can write the pattern for just one row and by using row compression, our 96 by 32 array of individual LEDs can be represented as an array of only 96 by 4 LEDs. To start, it is good form to pre-allocate a block of space for MATLAB to work with. We will call the 4-D matrix Pats.

Pats = zeros(4, 96, pattern.x\_num, pattern.y\_num); % Preallocate space

Next, we have to populate our blank pattern. The repmat function comes in handy here, typing help repmat in the command window gives a description of how it works. We start by populating the first frame (where x and y both = 1).

Pats(:,:,1,1) = [repmat([7\*ones(4,4), 0\*ones(4,4)], 1, 6),...

 repmat([0\*ones(4,4), 7\*ones(4,4)], 1, 6)];

Now we manipulate this first frame to populate the other channels. You may first want to do the math to convince yourself you have filled up all 96 LED columns.

for i = 1:pattern.y\_num

 for j = 2:pattern.x\_num

 Pats(:,:,j,i) = simple\_expansion(Pats(:,:,j-1,i), 49,96);

 end

end

for g = 1:pattern.y\_num;

 for i = 1:pattern.x\_num

 Pats(:,:,i,g) = circshift(Pats(:,:,i,1),[0 i]);

 end

end

Go through these lines to understand how they populate the pattern file by first creating a filled pattern with all identical X channel expansion, and then shift the matrix one column per each Y index. Be sure to note there need not be 96 Y channels. Because the experiment only wanted to test three different starting points of expansion, the Y channels could be individually assigned to three different starting points any number of ways (one of which is increasing the shift of ShiftMatrix). Just know this is not the only way to make this pattern, or any pattern!.

pattern.Pats = Pats;

new\_controller\_48\_panel\_map = [12 8 4 11 7 3 10 6 2 9 5 1;…

 24 20 16 23 19 15 22 18 14 21 17 13;…

 36 32 28 35 31 27 34 30 26 33 29 25;…

 48 44 40 47 43 39 46 42 38 45 41 37];

pattern.Panel\_map = new\_controller\_48\_panel\_map;

pattern.BitMapIndex = process\_panel\_map(pattern);

pattern.data = make\_pattern\_vector(pattern);

directory\_name = ‘C:\MatlabRoot\Panels\Patterns’;

str = [directory\_name '\Pattern\_expansion']

save(str, 'pattern');

The Panel\_map attribute of the structure is set equal to a map of the arena panels. The given matrix is an optimized arrangement of panel IDs, corresponding to the 48Panel\_4Bus arena configuration file (see below, or in appendix). Be sure that the name you give your pattern begins with “Pattern\_”. Run this m-file in MATLAB to create a .mat pattern.

To play this pattern, select from the PControl gui configurations > play pattern > choose your pattern, noting the majority of the frames are black (unpopulated). We will next use a number of functions created to ease pattern creation, all of which are included in the \Panels\Pattern\_tools\ folder you already included in the MATLAB path. Refer to the help, or directly to the functions for better understanding.

Understand that Pattern\_expansion, when played with positive gain, will be expansion, but when played with a negative gain, will be contraction. Instead of needing to make a second pattern, we can use the one pattern twice for our desired experiment.

Making an experiment script

In order to have more control over patterns and settings, an experiment script will be needed. There are very few necessary settings (the same ones from the PControl GUI), and they can be changed easily using the Panel\_com command. To run an experiment these are the only lines needed:

Panel\_com('set\_mode',Mode);

Panel\_com('send\_gain\_bias',[Gain\_X Bias\_X Gain\_Y Bias\_Y]);

Panel\_com('set\_pattern\_id', Pattern\_ID);

Panel\_com('set\_position', [Ind\_X Ind\_Y]);

Panel\_com('start')

pause(Time) % The pattern will run for this ‘Time’

Panel\_com('stop')

Each of the Panel\_com commands are explained in the Software Overview section and in the appendix. Some example values for each of the variables passed are as follows:

Mode = [0 0]; % Corresponds to closed loop for X and Y

Gain\_X = 1;

Bias\_X = 0;

Gain\_Y = 0;

Bias\_Y = 0;

Pattern\_ID = 1; % The first pattern on the SD card

Time = 3;

Ind\_X = 49; % This should center the pattern

Ind\_Y = 1;

If this were saved as an experiment script it would run a pretty boring experiment, doing nothing more than what you can from the PControl GUI. What makes a more exciting experiment are more conditions, each of which may have different Gains, Biases, Patterns, X and Y starting positions, and, importantly, Modes. An easy way to do this is to store all of the relevant condition attributes in a MATLAB structure called condition. Here is an example of how this is done where the conditions differ in three ways: speed of pattern (by changing Gain\_X), direction of pattern (by changing the sign of Gain\_X) and starting orientation of the pattern (by changing Ind\_Y). Remember our desired experiment starts expansion or contraction at one of three spots, and that by changing the direction (sign) of pattern, we switch from expansion to contraction.

speeds = [32 64 96];

time = 3;

condition\_num = 1; % The first condition value

for i = 1:length(speeds)

 for k = 1:2 % Positive and Negative Speed

 for g = [28 36 49] % 3 Starting positions corresponding to the 3 initial expansions or contraction points

 condition(condition\_num).Y\_ind = g;

 condition(condition\_num).pattern = 1; condition(condition\_num).X\_ind = 1;

 condition(condition\_num).X\_gain = speeds(i);

 condition(condition\_num).Y\_gain = 0;

 condition(condition\_num).X\_bias = 0;

 condition(condition\_num).Y\_bias = 0;

 condition(condition\_num).mode = [0 0];

 condition(condition\_num).time = time;

 if k == 1; % Set the value to be pos

 condition(condition\_num).X\_gain = condition(condition\_num).X\_gain;

 else

 condition(condition\_num).X\_gain = -condition(condition\_num).X\_gain;

 end

 condition\_num = condition\_num + 1;

 end

 end

end

num\_conditions = condition\_num - 1;

These for loops create the condition structure with 18 different conditions, the value of the num\_conditions variable. Convince yourself by examining the structure in MATLAB, that all conditions were made and stored.

Now, by using the length of the num\_conditions variable, we can loop through the different conditions by adding a for loop and some other details to the first code of this section. There is one critical component missing, a voltage encoded condition signal. In order for us to tell each condition apart when recording the experiment, each condition must be associated with some signal also recorded. Thankfully the data acquisition toolbox makes this painless, we create an analog voltage object that can encode the condition signals using a value of 1-4 volts. This example is for a standard NIDAQ board with 32-bit Windows, see below and MATLAB help for instructions on making analog output objects with other boards and OS versions.

AO = analogoutput('nidaq', 'Dev1');

chans = addchannel(AO, [0]);

Each time we switch conditions, the value of condition\_num changes to encode the next condition. The command putsample from the data acquisition toolbox allows this.

putsample(AO, condition\_num/(num\_conditions/4))

That is the last component of the script. Here it is, from start to finish, with a few simple lines added to make it run nicely.

clear all

%% Make the condition structure

speeds = [32 64 96];

time = 3;

condition\_num = 1; % The first condition value

for i = 1:length(speeds)

 for k = 1:2 % Positive and Negative Speed

 for g = 1:3 % 3 Starting positions

 condition(condition\_num).Y\_ind = g;

 condition(condition\_num).X\_ind = 1;

 condition(condition\_num).pattern = 1;

 condition(condition\_num).X\_gain = speeds(i);

 condition(condition\_num).Y\_gain = 0;

 condition(condition\_num).X\_bias = 0;

 condition(condition\_num).Y\_bias = 0;

 condition(condition\_num).mode = [0 0];

 condition(condition\_num).time = time;

 if k == 1; % Set the value to be pos

 condition(condition\_num).X\_gain = condition(condition\_num).X\_gain;

 else

 condition(condition\_num).X\_gain = -condition(condition\_num).X\_gain;

 end

 condition\_num = condition\_num + 1;

 end

 end

end

num\_conditions = condition\_num - 1;

%% Create an Analog Output Object (AO)

AO = analogoutput('nidaq', 'Dev1');

chans = addchannel(AO, [0]);

%% Experiment

condition\_num = 1; % set this again to 1

num\_reps = 3; % define the number of repititions you want

fprintf('Trial beginning')

conds\_to\_run = randperm(num\_conditions);

fprintf(strcat('conds2run =', num2str(conds\_to\_run), ' \n'));

for i = 1:num\_reps

 for j = 1:length(conds\_to\_run) % for each different speed

 cond\_num = conds\_to\_run(j); % and take the values out of the condition struct

 Gain\_X = condition(condition\_num).X\_gain;

 Gain\_Y = condition(condition\_num).Y\_gain;

 Bias\_Y = condition(condition\_num).Y\_bias;

 Bias\_X = condition(condition\_num).X\_bias;

 Ind\_Y = condition(condition\_num).Y\_ind;

 Ind\_X = condition(condition\_num).X\_ind;

 Pattern\_ID = condition(condition\_num).pattern;

 Time = condition(condition\_num).time;

 Mode = condition(condition\_num).mode;

 fprintf('round %d, run %2d, cond num = %2d, speed = %2d, pause = %2d, start Y pos = %2d \n',i, j, cond\_num, Gain\_X, Time, Ind\_Y);

 disp('...next')

% Open Loop Begins

 % scale condition number to fit in 0-4V range

putsample(AO, cond\_num/(num\_conditions/4)) Panel\_com('set\_mode', [Mode]); % set to open loop with Panel\_com

Panel\_com('set\_pattern\_id', Pattern\_ID); Panel\_com('send\_gain\_bias',[Gain\_X Bias\_Y Gain\_Y Bias\_Y]); Panel\_com('set\_position', [Ind\_X Ind\_Y]); Panel\_com('start')

 pause(Time)

 Panel\_com('stop')

 end

end

clear AO

Important note: there are many other possible ways to set up an experiment script, here are a few:

Alternate closed loop ‘reward’ stimuli with open loop experimental stimuli. This helps to keep the fly flying and engaged

Make the experiment script into a function that can take a number of individual stimuli as input. This is useful if some conditions did not run properly during the first repetitions.

Acquiring the Data

Once the experiment script is working, and flies are ready to be run (see the Fly Preparation section), experimental data must be acquired. Data is usually acquired using a standard data acquisition board (DAQ), and a program such as LabVIEW, from National Instruments, axoscope, if you are using an Axon board, or the freeware Spikehound, developed by Gus Lott. The data acquisition toolbox in MATLAB works with all major DAQ boards, can be used in place of the GUI programs above, and has extensive help files for getting started. The programs require configuration to acquire all relevant data from the DAQ board, this is usually painless, but care should be taken to note the sampling rate each experiment is conducted at.

A typical experiment will start with some closed loop time to properly align the fly in the arena, starting the DAQ program, and then running the experiment script. If the fly stops flying during one condition, it is typically rerun before ending the DAQ program.

More detailed data acquisition instructions are available on the bitbucket sites listed in the introduction.

Preparing the Data

Depending on your DAQ program of choice you will now have a VI file from LabView, an abf file from axoscope, or a daq file from the Data Acquisition toolbox (i.e. Spikehound). Each of these formats can be loaded to the MATLAB environment with varying degrees of difficulty. In this example we will use a daq file, as this is the data acquisition native format. To read a daq file use:

Data = daqread('Filename.daq');

Note that for the proprietary axoscope format (abf files), reading into MATLAB requires a special function freely available on the MATLAB central file exchange website (www.mathworks.com/matlabcentral/fileexchange/) .This will import the file to a matrix named Data in the workspace. You can break the Data matrix into individual DAQ channels with the following strategy (if you are using the default configuration):

Left = Data(:,1); % left wing beat amplitude

Right = Data(:,2); % right wing beat amplitude

% Insert the rest of the channels here

num\_conditions = 18;

condition\_signal = round((num\_conditions/4)\* Data(:,6));

If your condition signal, or left and right wing beat amplitudes were recorded from different channels, then change the column extracted for each attribute. We must now break up the data in terms of condition, we have already recovered each condition signal in the condition\_signal variable, but there are some steps required to break the data up effectively. Here is the entire script necessary to do so, note this will not fit all situations, but is representative of voltage decoding strategies. The code is heavily commented, and should be relatively transparent after some thought.

Data = daqread('Filename.daq');

Left = Data(:,1); % left wing beat amplitude

Right = Data(:,2); % right wing beat amplitude

% Insert the rest of the channels here, also needs some data from the

% experiment to be explicitly listed for later use

% From the original condition struct

num\_conditions = 18;

Time = 3;

% Noted from the DAQ program settings

sample\_rate = 1000;

% Recover the condition signal for the entire column of analog readings

condition\_signal = round((num\_conditions/4)\* Data(:,6));

% establish what is a significant difference between condition signals, in

% this case 85% of the difference between two sequentials is good enough

sig\_diff = (0.85\*(num\_condtions/4));

% create a variable with all of the Positions where the condition signal is

% significantly different (defined above)

diff\_Pos = (find( abs(diff(condition\_signal)) > sig\_diff));

max\_Pos = length(diff\_Pos);

% create two arrays to define blocks of data, and make the current Position

% in the data sweep two

start\_Pos = []; end\_Pos = []; cur\_Pos = 2;

% Search through the positions to establish blocks of data for each

% condition using both the diff\_Pos and the time passing between each set

% of conditions to verify a condition change.

while cur\_Pos <= max\_Pos %when the current Position is less than or equal to the maximum

 Pos\_diff = (diff\_Pos(cur\_Pos) - diff\_Pos(cur\_Pos - 1));

 range\_start = diff\_Pos(cur\_Pos - 1);

 range\_end = diff\_Pos(cur\_Pos);

 if ((Pos\_diff > (Time-.15)\*samp\_rate)&&(Pos\_diff < (Time+.15)\*samp\_rate))

 start\_Pos = [start\_Pos diff\_Pos(cur\_Pos - 1)];

 end\_Pos = [end\_Pos diff\_Pos(cur\_Pos)];

 end

 cur\_Pos = cur\_Pos + 1;

end

OL\_data\_length = Time\*sample\_rate;

% Create an empty struct, OL\_Data for each condition and data line

for j = 1:num\_conditions

 OL\_Data(j).Left = [];

 OL\_Data(j).Right = [];

end

% Populate the OL\_Data struct for each condition, Index will correspond to

% each condition number in the original condition struct

for j = 1:(length(start\_Pos))

 curr\_Range = start\_pos(j):start\_pos(j)+OL\_data\_length-1;

 % make a condition number for each data range that corresponds to the

 % actual condition number

 cond(j) = round( mean(condition\_signal(curr\_Range)));

 Index = cond(j);

 OL\_Data(Index).Left = [OL\_Data(Index).Left; Left(curr\_Range)'];

 OL\_Data(Index).Right = [OL\_Data(Index).Right; Right(curr\_Range)'];

end

% cond and OL\_Data from the previous for loop need to be saved together, we

% will use a struct named fly

fly.condition = cond;

fly.OL\_Data = OL\_Data;

save fly fly

You now have your data broken up into several conditions ready to be graphed! More data, such as the frame index, and the wing beat frequency could be added to this data set as well.

This brings us to the end of the first sample experiment tutorial; more information is available on the bitbucket sites.