**Panels User Guide**

Document History:

Revisions 1-3 (Theory and Practice of Insect Flight Simulators)

Mark Frye, Michael Reiser, Michael Dickinson

Revision 4 (Panels User Guide)

Stephen Holtz, Chuntao Dan, Sung Soo Kim

**Introduction**

Flight arenas are a powerful tool for addressing questions about sensory control of behavior and when paired with genetic techniques an equally powerful tool for probing neural circuits. This document offers an introduction to the flight arena and should serve as a reference when starting to run experiments. It contains a brief history, an overview of some of the physical components on the arena, and documentation on arena operation from a computer (see the outline below). Troubleshooting, assembly and required downloadable content is all available on the Panels bitbucket website (<https://bitbucket.org/mreiser/panels/>) or the sister bitbucket site, Panels Hardware (<https://bitbucket.org/mreiser/panels-hardware/>). Both websites are open to the public. Throughout the document we tend to refer to the tiled LED array as the “arena”, composed of individual “panels”, the image they display as “patterns”, and the black box that gives it all life the “controller”.

**Outline**

The rest of the documentation will be divided into the following sections:

History and Principles of Operation – Background and principles of arena operation.

Fly Preparation – Fly housing and tethering with some helpful tips.

Hardware Overview – The common physical components of an arena.

Software Operation – Detailing the components of software for experiments.

Example Experiments – Using a simple tutorial.

Technical Appendices – For more software specific commands and operation

**History and Principles of Operation**

The devices and techniques described in this document owe much to approaches developed at the Max Planck Institute for Biological Cybernetics in Tübingen, Germany. This institute grew in the early 1960s around a group of four scientists: Werner Reichardt, Karl Götz, Valentino Braitenberg, and Kuno Kirschfeld, who adopted the visual system of flies as a general model system for neural processing. The term ‘Biological Cybernetics’ was used to describe to use of information theory, computer science, control theory, etc. to characterize biological systems- the term ‘reverse engineering’ is often used today to describe similar approaches. This enterprise generated a remarkable body of work, which ranged from detailed anatomical descriptions of visual system circuitry (e.g. discovery of ‘neural superposition’ in the fly) to sophisticated behavioral and physiological measurements (e.g. characterization of the lobula plate tangential cell system) to development of mathematical model of neural processing (e.g. refinement of the Hassenstein-Reichardt elementary movement detector model).

Perhaps less appreciated, was the work of Karl Götz, who developed an ingenious set of experimental methods for studying the optomotor control of flight in the fruit fly, *Drosophila melanogaster*. The work influenced not only the techniques described here, but also served as the direct foundation for many recent studies of visual spatial memory in *Drosophila*. The focus of his pioneering work was to produce visual stimuli with specific spatial and temporal properties, record the resultant behavioral responses, and then derive the intervening transfer functions. By formalizing the functional relationships between visual input and motor output, he was able to develop and test specific hypotheses about the underlying neural structures. The work was made enormously challenging because there was no “off the shelf” instrumentation at all for such experiments. Götz literally drew the plans for the devices he used throughout his career, often leaving them on the desk of the chief institute machinist before walking home in the early hours of the morning. Götz spent equal time developing ingenious ways of presenting visual stimuli to flies, and clever ways of recording, quantitatively, how the flies responded. One early device was a complex braid of fiber optic filaments that took the images displayed on an analog oscilloscope and wrapped them into a cylinder surrounding the fly. Another device employed active electromechanical feedback to record the yaw torque generated by a tethered fly. One of his more ingenious experimental techniques was a system that optically tracked the beating wings of a tethered fly, and electronically coupled these signals to rotation of a spinning glass disk, such that light projected through fish eye optics created a pattern of stripes and spots on a circular cylinder surrounding the fly. As the fly (suspended in the middle of a rotating striped drum) tried to turn left, bilateral changes in wing stroke amplitude triggered servomotors to rotate the drum to the right. Under these conditions, the fly has “closed-loop” control over the visual panorama. Using such systems, Götz pioneered studies that linked the responses of the flies visual system to its motor responses in the form of muscle activity, wing motion, and aerodynamic forces.

Recent technical advances have made flight simulators easier to build and more sophisticated, but the basic architecture remains: present a visual stimulus to the fly, record what it does (in open loop), or provide it with the opportunity of changing what it sees (in closed loop). The new systems are modular and fully programmable for visual images of varying brightness, contrast, and spatial layout.

**Fly Preparation**

Housing Flies

For normal, wild-type experiments, flies are generally stored in incubators at fixed temperatures on a regular light cycle. For example flies may be stored at 25 C with lights on from midnight to 4 pm. Following are some guidelines and tips on storing flies.

* Flies are crepuscular animals with activity peaks in the early morning and late afternoon – when it is cool enough to fly and light enough to see. Keep your flies on a strict light:dark cycle and perform your experiments during their crepuscular peaks. The afternoon peak is typically the most convenient. Woe to those that think such details don’t matter.
* It is quite helpful to use flies reared at low density in bottles, as opposed to high density in vials, as is unfortunately standard in genetics labs. Presorting flies a day or two before running the experiment seems to help flight time in some stocks.
* Laboratory stocks which have reproduced for some time in vials can, over time, begin to fly less reliably than freshly caught and established lines. If possible, crossing two lab stocks can results in very high performance flies.

Tethering Flies

The most critical step in collecting high quality data in a flight simulator is tethering the animals. Well-chosen, healthy, properly-tethered (i.e. ‘happy’) flies will generate hours of data each day. Crappy flies (*sensu* Reiser), poorly tethered are not worth putting in the arena.

Once you have chosen a healthy batch of flies (see Housing Flies and Effectively Using Flies), the next step is to tether them. In brief, a batch of flies is transferred from a bottle to a small vial that is inserted in a brass block on a cooling stage set at about 4 degrees C. The low temperature anesthetizes the flies within 30 seconds or so. The flies may then be sprinkled on the surface of the cooling stage. Choose a large, healthy-looking fly with wings nicely folded back over its abdomen for transfer to the socket of the ‘sarcophagus’ using a suction wand. The trickiest part of the process is coaxing a fly into proper position within the sarcophagus using a fine brush. The fly is then held gently in place by suction applied through holes in the bottom of the sarcophagus. The fly is then tethered to a fine tungsten pin using UV-activated glue. Here are some helpful tips on the tethering: Here are a few hints regarding the choice and care of flies in preparation for tethering:

* Avoid damaging the wings with the brush, or the suction sources.
* Avoid getting water on the fly wings or in the fly esophagus by laying down a kimwipe on the cooling stage and thoroughly wiping the surface of the sarcophagus.
* Use only enough suction in the sarcophagus to stabilize the fly, the suction should not be so strong that the wings are pulled down off the back of the fly. The shorter time the fly is under suction, the less condensation will accumulate in the sarcophagus.
* Use an etched tungsten probe with a loop at the end to apply a droplet of glue to the tethering pin, then use the 3-D manipulator to transfer the glue from the pin to the fly.
* Alternatively, the glue can be dropped onto a glass slide and the very tip of the tethering pin may be dipped in. This forms a small droplet at the end of the tether.
* The most common rookie error is using too much glue. Basically, you should use the bare smallest amount that is required to hold the fly. When you are learning to tether, if you aren’t losing an occasional fly because it breaks free of the tether, you are probably using too much glue.
* When gluing the head, make sure that it is oriented straight ahead, and that the head-neck are in a neutral posture, not craned up or down.
* You want the final angle between the tether pin and the long axis of the fly (the pitch angle) to be 90 degrees. This is quite critical. If the pin leans too far back, the wings will hit it when they flap; too far forward and the fly’s stroke plane will not be horizontal in the flight arena as required for the optics of the wingbeat analyzer. Improper pitch alignment is the second most common rookie error.
* Make sure that the pin is bilaterally centered on the thorax, any misalignment in body roll makes the prep unusable for focusing the wingbeat analyzer.
* Hit both sides of the glue with a 10 second burst of UV light, get the tip of the gun as close to the fly as possible, but be careful not to bump the prep.
* Don’t forget to relieve the sarcophagus suction before removing the tethered fly. This is another common rookie error.
* Watch through the microscope as you withdraw the fly from the sarcophagus – she may stick, and pull free from the tether – if you’re watching you can correct by adding a bit more glue. Be familiar with the manipulator so that you pull the tethered fly up, instead of accidentally skewering it on its own tether.

Effectively Using Flies

Once tethered properly, there are many other tips to get the most out of your flies during experiments:

* Animals that are 2-3 days old generally provide the best data, but older flies will work if they are healthy.
* Females fly longer and more readily than males, but this may simply be a scaling effect – female flies are much larger and thus contain greater energy reserves. Large gravid females (with big white abdomens) seem to be the most behaviorally robust. A good, healthy 1.2 mg female will fly for about 1 hour, depending on the experiment.
* It may be helpful to starve flies for 4-8 hours prior to experiments. Starved flies tend to fly more. When starving, provide a damp kimwipe in the vial or bottle so that the flies do not desiccate.
* If necessary for an experiment requiring repeated measures, it is possible to revive a spent fly by feeding it sucrose solution. It is then best to wait at least half an hour to give it chance to process the sucrose into trehalose. Again, your fed, tethered fly may be reluctant to fly again until it is off food for several hours.
* Temperature is flies’ kryptonite. They prefer cool temperature for flight (c. 15-20 degrees C). Unlike moths and bees these animals do NOT need to warm up their flight apparatus and will get quite squirrelly at temperatures over 25 degrees C. This is often a problem because the arena display generates heat. The best setting for experiments is a cool dark room.
* If possible humidity in the experimental room should be at least 50% to prevent fly desiccation and encourage flight.
* To prevent tiring during starving or while waiting to use a fly in an experiment, a small (size of the fly) piece of paper may be placed on the tethered animal. This is most practically done when the fly is held fixed by its tether in a tether holding device.
* A convenient tool for holding flies on tethers while not using them can be cut out of plastic, but a Styrofoam block weighted at the bottom suffices.

**Hardware Overview**

The components of a flight arena setup consist of: An LED arena, an Optical Wingbeat Analyzer, an arena controller, a data acquisition board, an oscilloscope and a computer. In this overview, the optical wingbeat analyzer, controller, arena and oscilloscope will be discussed briefly. The detailed descriptions of the components and instructions for configuration are available on the bitbucket websites listed in the introduction.

Oscilloscope

An oscilloscope is an essential tool in the flight arena for properly aligning flies. Details on the setup are available on the bitbucket websites. Just note here that in order to view useful wingbeat waveforms from the optical wingbeat analyzer, this must be properly configured as well.

Optical Wingbeat Analyzer

The sensor of the wingbeat analyzer (WBA) is composed of two infrared-sensitive silicon wafers (shown below in *red*, one for each wing). An infrared LED suspended above the fly casts a shadow of the beating wings onto the sensor. An optical mask and a high gain amplifier circuit condition the sensor signals such that the final output is a voltage proportional to the position of the shadow cast by each beating wing. Increasing voltage represents increasing forward excursion of the wing, and therefore larger stroke amplitude.

Viewed from above (fly facing up the page), the shadow cast by the wings (shown in *gray*) must be laterally centered over the cutaway mask (dashed outline), a bit behind the forward edge. The size of the shadow is also important and can be adjusted by (1) moving the fly vertically, (2) moving the IR wand vertically, (3) moving the sensor surface vertically. Once these dimensional adjustments have been optimized, you should maintain them from experiment to experiment by placing the flies in the same position.

**good focus**  **too far forward, too large** **too far back, off to one side**

envelope of stroke amplitude

Outline

of mask

ventral flip

The IR wing sensor provides the analog signal to the wingbeat analyzer, which in turn detects the frequency and amplitude of the downstroke-upstroke reversal, also called the ventral flip. Proper fly alignment over the sensor is crucial – the wingbeat analyzer is a robust instrument and will report spurious values even if the input signal is messy.

If the fly is properly focused over the sensor, then the resultant signal from both the left and right outputs of the wingbeat analyzer should look like this *for each wing stroke*:

* Narrow waveform
* Looks like a little hat = = “hütchens”
* quantal cycle-by-cycle amplitude “pops”
* Second peak larger than first

Good hütchens!

* Broad waveform
* No cycle-by-cycle amplitude variation
* Saturated wing signal

Bad hütchens, more like a lüdenhut (ask MD) fly is likely too far forward over the mask

* Narrow waveform
* “Quantal” cycle-by-cycle amplitude variation
* First and second peak equal height, or second is smaller

Bad hutschienes, fly is either too far away from the sensor surface, or too far from IR source, or both

Changes to the hutchens that come from moving the fly in the arena are best learned by experimentation. If possible, align the fly while presenting a closed loop stimulus (see below) such as a dark stripe for the fly to fixate on. Some general guidelines for aligning the fly are as follows:

* If there is no amplitude variation (the hutchens are not ‘bouncy’) the fly is likely too far forward. If the amplitude seems to die down too far, increasing the current from the IR LED or moving the IR LED closer to the fly may help.
* If the left and right hutchens are not identical in shape, double check that the fly is glued symmetrically, is facing straight forward, and is centered under the LED and above the IR sensor. This can be difficult, but well worth the time spent, as it avoids later frustrations!
* If the wing hutchens disappear altogether or are saturating the wing beat analyzer, the fly is either too close to the IR LED and forward, or too far back and away from the IR LED, respectively.
* If fixing a stripe seems to be an issue for a (wild type) fly, adjusting left or right slowly and waiting for better fixation is probably necessary. By backing the fly up, or reducing the current from the IR LED, you can also reduce the difficulty of fixating. To ensure proper alignment, these settings should not be different than those used in actual experiments.

The WBA tracks the analog wing sensor signal voltage, and measures a suite of parameters for each wing stroke (defined by the inflections in the hütchens occurring between the user-defined trigger and gate values):

Inputs

* **Source**: left or right wing to set Gate and Trigger
* **Trigger**: voltage threshold to detect peak of downstroke
* **Gate**: time frame to detect peak of downstroke
* **Gain**: amplification of hutschienes – should read 2.5-3.5 Volts for standard DAQ
* **Filter**: low-pass filter analog sensor signals

Outputs

* **Left**: analog signal from IR wing sensor
* **Right**: analog signal from IR wing sensor
* **Frequency**: stroke frequency in cycles/sec
* **L-R**: left minus right amplitude - proportional to yaw torque
* **L+R**: left plus right amplitude - proportional to thrust
* **Flip**: a brief TTL pulse synchronized with the ventral flip
* **Sync**: a TTL pulse synchronized with each wing beat - used to trigger an oscilloscope sweep

Arena

LED flight arena: a modular array of 8x8 dot matrix LED panels. Each panel is independently addressable – i.e. can show a different visual pattern and can display 8 intensity levels (grayscale). In the current configuration, each LED (pixel) subtends (no more than) ~3.5 degrees at the retina. The arena should always be powered-up when the controller power switch is toggled.

Controller

The controller is the interface between the PC and the arena. There are two versions of the controller the newer, slimmer version is 3.0, some details of the controller ports are listed below.

Arena controller v 3.0 with Newest Controller Code

* ADC0: analog input in mode 1 and 2 of channel x
  + This input should be in the form of L-R. That is, with a negative gain in modes 1 and 2, a negative signal will cause a decrease in channel frame and a positive signal will cause an increase in channel frame number.
* ADC1 : analog input in mode 1 and 2 of channel y
  + This works in the same manner as ADC0
* ADC2: analog input in mode 3 of channel x
* ADC3: analog input in mode 3 of channel y
* DAC0: update current frame number (in the unit of volt) in mode 1, 2, 3, 4, and PC dumping mode of channel x, update analog output in mode 5 (debugging function generator) of channel x;
* DAC1: update current frame number (in the unit of volt) in mode 1,2,3, 4, and PC dumping mode of channel y , update analog output in mode 5 (debugging function generator) of channel y;
* DAC2: unused
* DAC3: unused
* Int0: unused;
* Int1: timing for fetching and displaying each frame when controller works in default mode and PC dumping mode. It is set to high before controller fetches a new frame data and reset to low when the controller finishes sending the data to the arena.
* Int2: laser trigger (see appendices for usage information)
* Int3: unused

Arena controller v 2.2 (‘black box’)

Power cycle –LED arena should be on before toggling the controller power (on or off)

* ADC1-4: inputs used to control the X and Y pattern position
* DAC1: voltage proportional to X Position
* DAC2: voltage proportional to Y Position
  + The DAC voltages will be values between 0V – 5V. The size of the voltage steps is set by the number of frames in X, Y for the current pattern (e.g. 96 frames in one channel would lead to voltage steps of 5/96 V; frame index 48 would be roughly 2.5 V). These voltage levels are remarkably consistent and can be used to recover the exact frame value for patterns of up to 500 frames (can only be done approximately for larger patterns).

**Software Operation**

Arena operation requires a MATLAB installation with a data acquisition toolbox license, the specific files necessary for interfacing with the controller can be downloaded from the bitbucket website and added to the MATLAB path. This section contains an overview of the GUI components and functions, an introduction to software control of the arena, pattern building commands, pattern acceleration discussion, and SD flash programming instructions. Many of these concepts are rehashed in the example experiment section.

MATLAB GUI Control: PControl

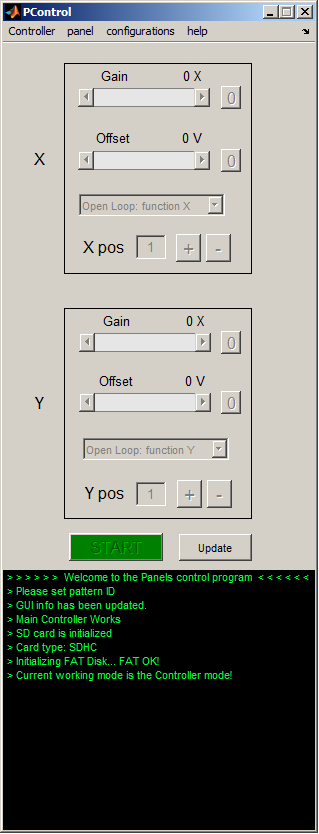
From the PControl GUI a lot of the arena functionality is accessible, it is a good place to start learning the system.

Quick Start

1. Switch on the arena. Panels will show addresses.
2. Insert pre-programmed SD flash card into the controller.
3. Switch on the controller. Verify that the left-hand green PWR LED is on steady. Arena will go dark.
4. Start Matlab.
5. In the Command Window, type “PControl”.
6. Verify the serial connection between the PC and the controller, select: Controller->blink LED, verify the 2nd red LED for “Memory status” blinking on the controller.
7. Load a pattern in PControl with the menu Configurations->set pattern ID -> Choose a Pattern from the dropdown menu.
8. First re-zero X and Y Gain and Offset. Set Gain to a non-zero value in the appropriate channel (X or Y), hit Start, and Stop

Open-Loop operation

Once you have loaded a pattern to display in the arena, set the menu options in the X and Y front panel drop down menus to ‘Open Loop: function X’, and ‘Open Loop: function Y’ (the default modes). Hit Start. Play with the Gain and Offset values at will. Can you figure out what X pos and Y pos do? Program a moving pattern by using the ‘functions’ menu to load periodic waveforms. Now manipulate the Gain and Offset controls and hit Start.

******

Open Loop with an external waveform

From the ‘patterns’ menu, load the checkerboard pattern (hopefully, it is Pattern Index 1) Connect the output of a function generator to ADC1. From the X drop down menu, select ‘Position: CH5 sets X ind’. Set the X Gain to 2.0 and hit Start. Vary the controls on the function generator, and verify that the pattern moves in register with function generator output. Move the checkerboard pattern horizontally by loading an internal function in the Y-channel and selecting the drop-down menu ‘Open Loop: function X’.

Closed-loop operation

The ‘closed-loop’ mode is similar to running the pattern in Open Loop with an external waveform. The only real difference is that the external function is a time-varying voltage proportional to the fly’s wing amplitude (i.e. steering torque). To give the fly control over the X pattern (e.g. load a single stripe pattern, which should occupy the X channel), plug the L and R Amplitude outputs from the Wingbeat Analyzer (not to be confused with L and R Signal Out) to ADC0 and ADC1 respectively. Set the X-channel drop-down menu to ‘Closed Loop: CH1-CH2’. Set the X Gain to 2.0, and hit Start. Use the X Offset to balance steering asymmetry. If you wish to run the Y pattern in closed-loop, connect L and R Amplitude to ADC2 and ADC3 and set the Y-channel drop-down menu to ‘Closed Loop: CH3-CH4’.

Introduction to software control from Matlab

PControl.m (controller GUI) frequently calls Panel\_com.m, a case structure of sub-functions used by the GUI. Anything the GUI can do may be executed on the command line (or in a script) with arguments to Panel\_com(‘argument’, [value]). You can think of ‘X’ and ‘Y’ as axes of the memory buffer that stores the individual frames to display on the panels. For the arenas we constructed here, each frame of the display will be a 96 x 32 pixel bitmap (corresponding to the number of individual LED’s around the azimuth and zenith, resp. of the arena). X and Y correspond to the two dimensions of the array of frames, they do NOT necessarily correspond to the coordinates of the display with respect to the fly. Here’s an example: consider a vertical stripe rotating 360 degrees around the fly. This pattern requires 96 individual frames, one for each column of pixels such that if they are displayed sequentially, the pattern looks like a stripe rotating smoothly around the yaw axis. The 96 frames can be stored in X(1:96), Y(1). By contrast, consider a rotating striped drum with each black-white pair composed of 8 pixels. The pattern can be stored in only 8 frames, X(1:8), Y(1), and simply iterated over and over to evoke the perception of pattern moving smoothly around the yaw axis.

In general, you use PControl and associated functions to design, build, and test the patterns for experiments. Then, you execute individual functions in scripts to conduct a controlled, repeatable experiment. For example, Panel\_com is a function with a series of different arguments – anything you do with PControl (and some things you cannot) may be programmed by Panel\_com on the command line. Here are some examples:

Panel\_com('set\_pattern\_id',4); % load pattern number 4 from compact flash

Panel\_com('set\_mode',[1,0]); % closed-loop mode “Closed Loop: CH1-CH2”

Panel\_com('set\_position'[49, 1]); % set the position to X=49,Y=1

Panel\_com('send\_gain\_bias'[12,0,0,0]); % send Gain and Bias values

Panel\_com('start'); % begin pattern motion

pause(20); % let the pattern run for 20 seconds

Panel\_com('stop'); % stop pattern motion

Panel\_com commands are detailed in Technical Appendix 1.

Building a pattern file

The display of visual patterns requires three separate steps: First, patterns must be created in numeric form most typically using a convenient program such as Matlab, and saved in a suitable format. Second, this file, containing the display information, is burned onto a flash memory card from a computer. Third, the flash memory card is inserted into the Arena Controller, and the contents are downloaded onto the panels using the Controller’s software (low-level, C code).

Although involving many steps, this process is straightforward, especially if using our custom Maltab functions that make this first step easier. The most complicated (for the user) part is constructing the arrays that contain the display information for the panels in the arena. In doing so it is important to keep several concepts in mind. The number of unique panels required for a display may vary.Each 8x8 pixel panel is given an identity, which is set using the procedure described above. When you power up the arena, these IDs are displayed on each panel. The identities may or may not be unique. For example, you may want an arena consisting of 12x4 panels, in which you toggle through a uniform display of grayscales. In this case every panel can have the same ID, because they will each display the identical 8x8 pixel pattern. In contrast, a standard stripe fixation experiment will require 12 unique panel IDs – enough to encode the motion of the stripe around the circle, although each column of 4 panels can have the same ID because they will always display the same pattern. In contrast, an experiment in which you control the azimuthal and elevation angle of a small spot will require that all 48 panels in your 12x4 cylinder have a unique ID. The important thing is to recognize that the way you create a pattern will depend upon the number of unique panels in your arena and their spatial distribution.

Why would you want non-unique panel IDs? Using unique panel IDs provides the most flexibility, but it does require larger patterns and those patterns will necessarily run slower – the controller must update the complete display. Any display system will have some upper bound on the rate at which it can update the display. This is an issue that should not be ignored, and in the cases where a particular experiment requires a very high frame rate, one option to consider is non-unique IDs. For example, a 12-panel pattern will achieve roughly 4 times the maximum frame rate as a 48-panel pattern.

Assuming you are making patterns in Matlab using the utility functions written by Michael Reiser, the critical step in creating a display file is using the ‘make\_pattern\_vector’ command, which operates on a ‘pattern’ structure. The pattern structure has eight critical fields that are required before the structure may be saved in an ‘\*.mat’ file and loaded on the flash card.

**pattern.x\_num** = the number of frames in the ‘x’ channel

**pattern.y\_num** = the number of frames in the ‘y’ channel

**pattern.num\_panels** = number of unique panel IDs required

**pattern.gs\_val** = grey scale value; must be either 1, 2, or 3.

‘1’ indicates all pixel values in pattern.Pats are binary (0 or 1)

‘2’ indicates all pixel values in pattern.Pats are either 0, 1, 2, or 3.

‘3’ indicates all pixel values in pattern.Pats are either 0, 1, 2, 3, 4, 5, 6, or 7.

**pattern.Pats** = This field contains the basic data for the displayed images.

Pats is a matrix of size (L,M,N,O), where:

L is the total number of pixel rows

M is the number of pixel columns

N is the number of frames in the ‘x’ direction, same as **pattern.x\_num**

O is the number of frames in the ‘y’ direction, same as **pattern.y\_num**

Thus, each entry in this LxMxNxO matrix is either one bit (for pattern.gs\_val =1) or 2 bits (for pattern.gs\_val =2), or 3 bits (for pattern.gs\_val =3). Most of the work in generating a display pattern is in creating the Pats matrix. However, if a simple azimuthal shift is required (as in a yaw control experiment), the function ‘ShiftMatrix.m’ can be used to create shifted maps required to depict the moving image.

**Pattern.Panel\_map** = this is a vector or matrix that encodes the orientation of the panel IDs.

See notes above for why this is important. A simple example might be:

Pattern.Panel\_map = [1 2 3 4 5 6 7 8 9 10 11 12];

For a case where columns of panels display the same information. A more complicated example might be:

A = 1:48;

pattern.Panel\_map = flipud(reshape(A, 4, 12));

% 4 8 12 16 20 24 28 32 36 40 44 48

% 3 7 11 15 19 23 27 31 35 39 43 47

% 2 6 10 14 18 22 26 30 34 38 42 46

% 1 5 9 13 17 21 25 29 33 37 41 45

where every panel in a 12x4 panel arena needs a unique address. Panels should be addressed starting from 1, and going up to **pattern.num\_panels.** This restriction could generate confusion if you don’t use all 12 columns. For example, if you use only 7 columns with 3 rows of panels, then you have two choices. First, you may use the 4x12 panel map shown above as if you’re using all 12 columns. But it makes the pattern size larger and the speed could be slower. Instead, you may use different panel map and organize your panels accordingly as shown below.

% 3 6 9 12 15 18 21

% 2 5 8 11 14 17 20

% 1 4 7 10 13 16 19

But, you cannot use a map shown below, which is a truncated version of the 4x12 map.

% 3 7 11 15 19 23 27

% 2 6 10 14 18 22 26

% 1 5 9 13 17 21 25

**pattern.BitMapIndex =** a structure that encodes the spatial panel IDs.

This field can be generated using the function, ‘process\_panel\_map.m’, which operates on pattern.Panel\_map, e.g. ‘pattern.BitMapIndex = process\_panel\_map(pattern);’

**pattern.data =** final vector format of the data.

The final vector form of the data that is ready for output can be created using ‘pattern.data =

make\_pattern\_vector(pattern);’ . The resulting structure is then saved as a \*.mat file.

Here is an example Matlab script that uses these functions to create a simple panel file for a stripe fixation experiment:

pattern.x\_num = 96; % There are 96 pixel around the display (12x8)

pattern.y\_num = 1; % There is no vertical motion; only one frame is needed

pattern.num\_panels = 12; % This is the number of unique Panel IDs required.

pattern.gs\_val = 1; % This pattern will be binary , so grey scale code is 1;

Pats = zeros(8, 96, pattern.x\_num, pattern.y\_num); %initializes the array with zeros

stripe\_pattern = [ones(8,88),zeros(8,8)]; %dark pixels for the stripe

Pats(:, :, 1, 1) = stripe\_pattern;

for j = 2:96 %use ShiftMatrixPats to rotate stripe image

Pats(:,:,j,1) = ShiftMatrix(Pats(:,:,j-1,1),1,'r','y');

end

pattern.Pats = Pats; % put data in structure

pattern.panel\_map = 1:1:12; % define panel structure vector

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

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

directory\_name = 'c:\matlabroot\Patterns';

str = [directory\_name '\Pattern\_example1'] % name must begin with ‘Pattern\_’

save(str, 'pattern');

Accelerating your patterns

Let’s face it, we all like speed. All the fancy things that one can do with the controller will usually incur a cost—the patterns will probably display slower. By slower, we mean that the controller will spend more time refreshing an individual frame, and thus to keep up with the expected display rate (either set in open-loop by the function generator, or in closed-loop by the fly), the controller will drop frames. Suppose that we want the controller to display a 10 frame pattern at 250 Hz, but the maximum achievable frame rate is 95 Hz. The figure on the right should give a rough idea of the actual sequence of displayed frames as compared to the desired sequence.

How do you know your maximum achievable rate? Benchmark your pattern. This can be done automatically using the controller—the results are sent out through PC2 serial port. A quick and dirty benchmark can be performed by manually increasing the display rate and observing when the skip-indicating LED (green one, next to the power on LED in the middle of the controller box) starts to blink.

How does one get more speed?

1. Simplify the pattern; either by not using grayscale, or by reducing the number of panels used.
2. Use row compression. A large number of patterns we might care to try consist of identical pattern data for all rows of the pattern. When this is the case, a row-compressed pattern will just consist of one row instead of 8. This results in a speedup factor of at least 5. To enable row compression, simply make a pattern with only one row per panel, and include the setting (pattern.row\_compression = 1;) in the pattern making script. See example below.
3. Use ‘identity’ compression. The general method outlined for making and sending patterns is simple (elegant, perhaps) but should strike no one as *optimal*. Many patterns contain large swaths of pixels that are simply on or off. One simple shortcut that has been implemented is to simply send the row-compressed version of the panel data for a panel that corresponds to a pattern piece that is all one value (works for grayscale too!). This feature is not used at pattern making time, but rather while the pattern is running. Identity compression can be enabled by invoking: Panel\_com('ident\_compress\_on') from MATLAB. Seems like a brilliant idea, why isn’t it just always on? This brings up an important lesson in compression—there are no one-size-fits-all solutions. Sure this methods speeds up the pattern containing a single stripe, but consider a striped drum pattern—all of the extra comparisons needed to determine if a particular panel’s piece of the pattern can be sent as a single row, will slow things down considerably and never find a compressible pattern patch.
4. Think outside the box…each pattern can probably be optimized in its own way.

Here is one more example Matlab script that creates a pattern. This one creates a grayscale sine wave grating pattern for a 48-panel display, and uses row compression. This script creates two versions of the patterns with different spatial frequencies, and stores them as Y index 1 and 2.

pattern.x\_num = 96; % There are 96 pixel around the display (12x8)

pattern.y\_num = 2; % two frames of Y, at 2 different spatial frequencies

pattern.num\_panels = 48; % This is the number of unique Panel IDs required.

pattern.gs\_val = 3; % This pattern will use 8 intensity levels

pattern.row\_compression = 1;

% size of each frame is 4x96, because of row compression.

Pats = zeros(4, 96, pattern.x\_num, pattern.y\_num); %initializes the array with zeros

% make grating patterns, periods are 120 and 60 degrees, using all 8 gscale values

Pats(:, :, 1, 1) = repmat(round(3.5\*(sin((6\*pi/96)\*[0:95])+1) ), 4, 1);

Pats(:, :, 1, 2) = repmat(round(3.5\*(sin((12\*pi/96)\*[0:95])+1) ), 4, 1);

for j = 2:96 %use ShiftMatrix to rotate stripe image

Pats(:,:,j,1) = ShiftMatrix(Pats(:,:,j-1,1),1,'r','y');

Pats(:,:,j,2) = ShiftMatrix(Pats(:,:,j-1,2),1,'r','y');

end

pattern.Pats = Pats; % put data in structure

A = 1:48; % define panel structure vector

pattern.Panel\_map = flipud(reshape(A, 4, 12));

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

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

directory\_name = 'c:\matlabroot\Patterns';

str = [directory\_name '\Pattern\_grating\_row\_comp'] % name must begin with ‘Pattern\_’

save(str, 'pattern');

Programming the SD Flash

1. Once you have a collection of patterns that you wish to display in the arena, put them all into C:\MatlabRoot\Panels\ Patterns\. (You may in fact put them any place you wish, but this is the default directory).
2. Plug a SD Flash card into a port, using a USB multicard reader. Verify that it appears as a drive letter. You may be prompted to format it, or given some error about the drive being inaccessible, ignore these warnings.
3. In Matlab, run PControl, and select “configurations->load pattern to SD”.
4. In the Pattern Selection window, press Add Folder, navigate and select the folder storing your pattern files.
   1. You may also individually add the patterns, if this becomes time consuming to achieve the correct order of patterns, consider naming each pattern as “Pattern\_##\_id” where ## is the number you wish the pattern to be and id is an identifying name. This will ensure the patterns in the folder selected are added in the correct order
5. Hit “OK”.
6. In the Pattern Selection Tool window, verify the patterns you want to store
7. Press “Make Image”.
8. Press “Format & Burn”.
9. DO NOT take out the CF card until you see a message confirming that the CF card writing is finished.
10. Make sure the controller is switched off.
11. Pull the CF from the drive, and plug it into the arena controller.
12. Make sure that the arena is powered on, then switch on the controller.

**Note**: to be able to access the patterns on a CF from the PControl GUI, you must be using the CF on the same machine on which it was programmed. If you load patterns onto a CF on machine A, then you may only access them on machine B by using the Panel\_com(‘set\_pattern\_id’) command.

**Example Experiments**

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. [need image!!]

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 (the default directory where patterns are looked for is C:\MatlabRoot\Panels\Patterns).

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 LEDsNow 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) = ShiftMatrix(Pats(:,:,i,1), g, 'r', 'y');

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.

Putting the pattern on an SD card

The SD card is what your patterns will be played from. Plug an SD card into the PC using a USB reader or a multicard reader. Be sure it shows up under Computer as a Removable Disk, and note the drive letter is generally E. In Matlab, run PControl, and select “configurations->load pattern to SD” then in the Pattern Selection window, press Add Folder, navigate and select the folder storing your pattern files. You may also individually add the patterns, if this becomes time consuming to achieve the correct order of patterns, consider naming each pattern as “Pattern\_##\_id” where ## is the number you wish the pattern to be and id is an identifying name. This will ensure the patterns in the folder selected are added in the correct order. Press “Make Image” then “Burn”. If you are prompted for the drive letter at any time, type it into the prompt and press OK.

The command window should tell you if the burn was successful, once this happens remove the SD card and place it in the controller. You may need to turn the controller off and on for it to recognize the card correctly. To test the pattern, open up PControl and select configurations > set pattern ID > your pattern (from the dropdown menu). Now you can use the PControl GUI to run your pattern by selecting values and pressing the start button.

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 Spikehound or the Data Acquisition toolbox. 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.

**Technical Appendix 1 – User guide to the Panel\_com command**

The Panel\_com command allows for communication from Matlab to the arena controller. The function takes 2 values – the command and the numerical arguments (when needed).

Note: It does not make sense to issue some of these commands while the controller is ‘going’ – i.e. updating frames. For example, if a ‘All off’ command is sent while the controller is updating frames – the panels will be off for maybe a few milliseconds – and then the frames will keep updating, so it might not be obvious that this command did anything.

The following commands are useful for writing scripts to run experiments:

**Start** – same as pushing the start button on the GUI, controller starts updating frames.

Arguments: none

Usage: Panel\_com(‘start’);

Special case of this command for camera trigger, usage: Panel\_com(‘start\_w\_trig’);

**Stop** – stops the controller, pattern will freeze on last displayed frame.

Arguments: none

Usage: Panel\_com(‘stop');

Special case of this command for camera trigger, usage: Panel\_com(‘stop\_w\_trig’);

**All off** – sets all LEDs to off.

Arguments: none

Usage: Panel\_com('all\_off');

**All on** – sets all LEDs to on.

Arguments: none

Usage: Panel\_com('all\_on');

**Set grey level** – sets all LEDs to a greyscale level from 0 to 7:

Arguments: none

Usage: Panel\_com('g\_level\_0'); % set all panels to grey level 0;

…..

Panel\_com( 'g\_level\_7'); % set all panels to grey level 7;

**Set pattern ID** – sets the ID of the pattern

Arguments: a single value from corresponding to the pattern number (0 - # pats)

Usage: Panel\_com('set\_pattern\_id', 3); % set to pattern 3

Note: if ID used is too large – controller will crash - Panel\_com doesn’t check this

**Set controller mode** – sets the mode for the controller’s X and Y channels

Arguments: 2 values to set the mode for X and Y channels. 0 – open loop, 1 – closed loop, 2 – both, closed loop plus function as bias, 3 – External input sets position, 4 – Internal function generator sets velocity/position, 5 – internal function generator debug mode.

Usage: Panel\_com(‘set\_mode’, [0 1]); % X to open loop, Y to closed loop.

**Set Pattern position** – sets the position of the pattern. Controller will send this frame to panels.

Arguments: 2 values to set the X and Y positions, must be between 0 and num\_x or num\_y.

Usage: Panel\_com('set\_position', [10 1]);

Note, Panel\_com subtracts 1 from each of these values, because in Matlab, 1 is used as the start index, and the controller uses 0. Also, if the position value is too large – frame will not be correct - Panel\_com doesn’t check this.

**Set gain and bias** – sets the gain and bias for the X and Y channels on the controller.

Arguments: 4 values, 1 each to set gain\_x, bias\_x, gain\_y, bias\_y. Values must be signed integers between -127 and +127. Because of this – gain values are multiplied by 10 and bias voltages are multiplied by 20.

Usage: Panel\_com('send\_gain\_bias', [10 -10 0 20]); % sets gain\_x = 1X, bias\_x = -0.5 V, gain\_y = 0, bias\_y = 1 V (check PControl to verify this).

**Laser on/off** – enables/disables the laser trigger. When this is on, it enables the controller outputs a trigger on DIO 1 and 2 (of opposite activation – one is high, the other is low), when the pattern is in a certain position, that is currently hard-coded in the controller.

Panel\_com('laser\_on'); OR Panel\_com('laser\_off');

**Identity compression on/off** – enables/disables a compression scheme on the controller. Panel\_com('ident\_compress\_on'); OR Panel\_com('ident\_compress\_off');

**Set trigger rate** – sets the rate (frequency) of the optional camera trigger.

Arguments: 1 value, that must be an integer between 0 and 255.

Usage: Panel\_com('set\_trigger\_rate', [100]); % sets the trigger to 100 fps.

**Send laser pattern** – Use the laser switch with a user set pattern.  
Arguments: Panel\_com('send\_laser\_pattern',pattern);

Pattern, a numerical array with 96 elements whose value is either 0 or 1.   
Usage: Panel\_com('send\_laser\_pattern' [ones(1,24),zeros(1,24),ones(1,24),zeros(1,24)]);

Commands useful for setup/debugging -

**Toggle the LED** – as a debugging measure, blinks the second green LED on the controller

Arguments: none

Panel\_com('led\_tog'); % toggles controller LED

**Controller** **reset** – useful for debugging, resets the controller

Arguments: none

Usage: Panel\_com('ctr\_reset') % resets the controller

**Benchmark pattern** - measures the average frame rate for the current pattern, spits results out on second serial ports – use in conjunction with Hyperterminal.

Arguments: none

Usage: Panel\_com('bench\_pattern');

**Reset panel** – resets a panel, upon reset, panels show their address

Arguments: single value, panel address: 1-127, or 0 for all panels.

Usage: Panel\_com('reset', 0); %reset all panels

**Display panel** **address** – has panel display its address

Arguments: single value, panel address: 1-127, or 0 for all panels.

Usage: Panel\_com('display', 2); %have panel #2 display its address

**ADC test** – test the analog inputs to the controller – follow instructions in pop-up window.

Arguments – ADC channel to test, a number between 0 and 7

Usage: Panel\_com(‘adc\_test’, 3); % test ADC channel 3

**DIO test** – test the digital I/O lines to the controller – follow instructions in pop-up window.

Arguments: ADC channel to measure the DIO lines, a number between 0 and 7

Usage: Panel\_com(‘adc\_test’, 5); % test DIO using ADC channel 5

**Set panel address** – sets the address of an individual panel.

Arguments: 2 values, the first for current address, the second for the desired address (must be 0 – 127).

Usage: Panel\_com(‘address’, [0 12]); % all panels addressed as 0, go to 12

Panel\_com(‘address’, [11 55]); % all panels addressed as 11, go to 55

Note: 0 is a reserved address – all panels respond to zero.

**Set position function id** – set the function generator to work in a position mode.

Arguments: 2 values, the first for selection of x or y (0 or 1 respectively), the second for the function id.

Usage: Panel\_com(‘set\_posfunc\_id’, [1 9]); % sets x function generator with function id 9

Panel\_com(‘set\_posfunc\_id’, [2 7]); % sets y function generator with function id 7

Note: See technical appendix 2 for more detail

**Set velocity function id** – set the function generator to work in a velocity mode.

Arguments: 2 values, the first for selection of x or y (0 or 1 respectively), the second for the function id.

Usage: Panel\_com(‘set\_velfunc\_id’, [1 9]); % sets x function generator with function id 9

Panel\_com(‘set\_velfunc\_id’, [2 7]); % sets y function generator with function id 7

Note: See technical appendix 2 for more detail

**Set frequency of function** – set the frequency of the function generator of x or y.

Arguments: takes a value between 0 and 500. Problems may arise if not using multiples of 50.

Usage: Panel\_com(‘set\_funcx\_freq’, 50); % sets the frequency of x function generator 50.

Panel\_com(‘set\_funcy\_freq’, 100); % sets the frequency of y function generator 100.

Note: See technical appendix 2 for more detail

**Send function** – send piece of a function to the contoller.

Arguments: 52 byte values (integers 0 – 255). Byte 1 is identifier, 1 for X, 2 for Y, 2nd byte is

the function segment number (these are 50 byte length segments), starting at zero. then 50 bytes of data. Function buffer is 1000 bytes – so 20 of these calls are needed to fill it.

Usage: best done with a for loop – this example is for channel X.

for j = 0:19

start\_addr = 1 + j\*50; end\_addr = start\_addr + 49;

Panel\_com('send\_function', [1 j scaled\_func(start\_addr:end\_addr)]);

end

Note: functions must be made of signed values, between -127 and + 127, and must be scaled, so that a value of 20 corresponds to 1V [e.g. scaled\_func = round(20.\*func); ].

**Show bus number –** displays the bus numbers in the arena.

Arguments: none

Usage: Panel\_com(‘show\_bus\_number’);

**Set arena config** – set the current arena configuration file, searches the SD card for available options.

Arguments: see usage

Usage: From the PControl GUI, configurations > set arena config.

This works similarly to setting the current pattern ID. A dropdown menu displays the configuration files to loaded on the SD card to select for use with the current arena.

**Load config to SD** – loads an arena configuration file to the SD card, the default directory is Panel\_Controller\arena\_config\ .

Arguments: see usage

Usage: From the PControl GUI, configurations > load config to SD

This works similarly to loading patterns and functions to SD card. Once configuration files (mat files) have been added to the Configuration List window by Add or Add folder, selecting ‘load config’ will write the files to an SD card. Progress will be displayed in the command window. Note that the same limitation with pattern writing applies for configuration files. Other computers will not be able to read these names in the PControl GUI.

**Edit arena config** – make or edit a set of arena addresses to save or load to an SD card.

Arguments: see usage

Usage: From the PControl GUI, configurations > edit arena config.

An interactive arena configuration table allows creation and saving of a custom panel arrangement. Because there are 4 busses and the ability to skip an address, each panel ID can be assigned one of five values, 0-4. As noted on the arena\_config table:

Channel 0 - No panel/skip this address

Channel 1 – assign to bus 0

Channel 2 – assign to bus 1

Channel 3 – assign to bus 2

Channel 4 – assign to bus 3

This configuration should be compatible with the map used to create patterns. For example, the assignments in the config file should be to numbers that exist in the pattern map.

**Version** – display the Panel Controller version number in an open PControl GUI.

Arguments: none

Usage: Panel\_com(‘get\_version’); From the PControl GUI, help > version

Technical Appendix 2 – **Operational Mode Details**

To control time properly, it is essential to have a clear understanding of the operational scheme:

Internal function generator is a vector of 1000 values that plays out over 20 seconds, corresponding to 50 samples/sec. Each value is a single byte, specified as a signed integer, so these can range from -127 to +127 (maybe 128). These values directly corresponds to the frame rate played out, so a 1000 value vector of 5’s will produce 5 fps, if the gain is set to 1X. The **gain** is also specified as a single byte, signed integer, so the GUI implements gains from -10 to + 10, by sending out 10 times this value – X gain of 0.5 is sent as gain\_x = 5, Y gain of -2.2 is gain\_y = -22, etc.

The **bias** is also represented as 1 signed single byte integer. This value implements a virtual voltage, in the GUI, 1V is represented as the number 20 (this is only so we can use the -127 to + 127 range), this is scaled by 2.5 on the controller.

Note that in the calculations shown below, these are all integer operations implemented in C, the main significance of this is that there is no remainder kept after division (e.g. 10/3 = 3).

Each channel can operate in a few different modes, the timing of pattern display is as follows:

**mode 0, open loop** – in this mode the internal function generator is used to set the display rate.

So the rate in open loop is calculated as (using X as the example):

X\_val = 2\*function\_X[function\_counter]; % take the current function counter value

X\_rate = ((X\_val\*gain\_x)/10 + 5\*bias\_x)/2;

The division by 2 is implemented for added precision, but is shown here to explain the exact rate calculation. This value gives the frame rate in fps.

Ex1 – instantaneous function generator value of 10 (the default), gain\_x = 1X, bias\_x = 0,

X\_rate = ( (20\*10)/10 + 5\*0)/2 = 10 fps

Ex2 - instantaneous function generator value of 20, gain\_x = -1.5X, bias\_x = 0.3 V,

X\_rate = ( (40\*-15)/10 + 5\*6)/2 = -15 fps (obviously this is just 15 fps in opposite direction).

**mode 1, closed loop** – in this mode the input voltage difference is used to set the rate, for channel X, this is CH1-CH2, for y, this is CH3-CH4. The a2d conversion digitizes 5 V with a 10 bit resolution, so 5 V maps to 1024, this range is too high, so ultimately we divide this by 4 (divide by 2 twice), so that 1 V maps to about 51 fps.

The rate is calculated as:

X\_val = (X\_ADC1 - X\_ADC2 )/2;

X\_rate = ((X\_val\*gain\_x)/10 + 5\*bias\_x)/2;

Ex 1 – instant. X\_ADC1 = 204 (1 V), X\_ADC2 = 307 (1.5V), gain\_x = 2X, bias\_x = 0.5 V,

X\_val = (204 – 307)/2 = -51

X\_rate = ( (-51\*20)/10 + 5\*10)/2 = -26 fps

Ex 2 - Same as above, but with gain\_x = 1X, bias\_x = 0.5V

X\_rate = ( (-51\*10)/10 + 5\*10)/2 = 0 fps, note that the bias has exactly cancelled out the 1/2 V input difference.

**mode 2, closed loop with bias** – this mode is basically a combination of modes 1 and 2.

The rate is calculated as:

X\_val = (X\_ADC1 - X\_ADC2 )/2;

X\_rate = ((X\_val\*gain\_x)/10 + 2\*function\_X[function\_counter] + 5\*bias\_x)/2;

**mode 3, position control mode** – this mode is used in an experiment where online control of the frame position is required by some other equipment rather than the Wing Beat Analyzer. You can dynamically determine the position of frame as the experiment goes on.

This mode uses ADC4 and ADC5 (in an older controller, they are ADC5 and ADC6) to determine the position of x and y frame respectively. To utilize the full capacity of voltage range that the controller can accept, it uses gain\_x and bias\_x in a very different way from mode0~2. Users must understand the detailed computation to correctly determine the input voltage.

The digitizer is a 10 bit digitizer, so 5V maps to 1024, in this mode we do not divide by a fixed value, rather we divide by the gain. For channel x, the calculation of the frame to display is as follows:

index\_x = X\_ADC5/gain\_x + bias\_x; // Note that the division drops remainder.

if (index\_x >= x\_num) {index\_x = x\_num - 1;} //check if too big, x\_num is # frames in x

if (index\_x <= 0) {index\_x = 0;} //or too small

Ex1 – inst. input voltage is 2 V, this will convert to (2/5)\*1023 = 409. (Integer division drops the remainder.)

If gain is 1X, this will divide by 10, so index\_x = 409/10 = 40.

If the pattern has a 40th frame, that will be displayed, otherwise it will display the last frame.

Ex2 – same 2 V input voltage, gain = 1.5X, bias = -0.5

index\_x = 409/15 + -10 = 27 – 10 = 17, again, if there is no frame 17 then the last frame will display.

Ex3 – input is 0 V, gain is anything, bias = 1V.

index\_x = 0/anything + 20 = 20.

For best resolution, the user should determine the gain and the bias so that the frame index covers the maximum span of 5V. Usually, the bias is set to zero to avoid confusion, and only gain is calculated. The following Matlab routine generates reasonable voltage values for each x\_frame index and the gain.

function index\_voltage\_map = index\_to\_voltage\_converter(x\_num, y\_num)

gx = round( 1000/x\_num ); % instead of 1023 for margin

if gx>100

gx=100;

elseif gx<1

gx=1;

end

gy = round( 1000/y\_num );

if gy>100

gy = 100;

elseif gy<1

gy=1;

end

index\_voltage\_map.x = idx2volt([0:x\_num-1],gx);

index\_voltage\_map.y = idx2volt([0:y\_num-1],gy);

index\_voltage\_map.gain\_x = gx;

index\_voltage\_map.bias\_x = 0;

index\_voltage\_map.gain\_y = gy;

index\_voltage\_map.bias\_y = 0;

function mx = idx2volt(x, g)

for xi=1:length(x)

n = x(xi);

m = 0;

%m = floor(m\*100)/100;

ok = 0;

mo=[];

while ~ok

m = m+0.001;

ttmp = floor(m/5\*1023\*10)/10/g; % 10V=4095, 5V limit, and X\_ADC1 = analogRead(4)/2; index\_x = X\_ADC1/gain\_x + bias\_x;

% if ttmp > n+0.4

% ok=1;

% mx(1,xi) = m;

% end

if floor(ttmp)==n

mo(end+1) = m;

elseif floor(ttmp)==n+1

ok=1;

mx(1,xi) = median(mo);

end

end

end

**mode 4, function-based position control mode** – this mode is very useful if you want to present a preset sequence of patterns. Theoretically, you may use both mode 3 and mode 4 for the same purpose, but mode 3 has an electrical noise issue (on the communication using BNC cables), when you use too many frames (>200 or 300), that could affect your experiment at the expense of online control capability. Mode 4 is different in the sense that you cannot control the pattern online, but you can use the maximum number of frames that the system allows without any frame jittering caused by electrical noise. Mode 4 uses the function generator to numerically specify the current position. Gain and bias settings are irrelevant.

The following Matlab code generates functions.

func = [0 1 2 3 4 5 6 7 …… 95 ]; % sweeps from the first frame to 96th frame.

% Note that the actual position depends on the initial position. See below.

save(‘position\_function\_my\_sweep’, 'func');

Note that the prefix ‘position\_function’ is used. You must use this prefix to correctly apply your function in the system. Also note that the variable name is ‘func’. If you use a different name, then the provided utility function cannot convert it to a proper bit stream.

Next, you open PControl and select the menu “load function to SD”. Add ‘position\_function\_my\_sweep’, make image and burn it to the SD card.

If you use PControl for your experiment, you can use this function by selecting ‘set position function’ in the menu, then pick mode 4 (Position: function X (or Y) sets ind.) in the drop down menu.

If you use Matlab command line, the following sequence of commands should be applied.

Freq = 50; % There is a report that Freq should be multiple of 50

% to avoid freezing.

xfid = 1; % the id of position function starts from 1.

% So if you have only one function, it should be 1.

% If you have more, it increment by 1.

yfid = 1; % Assuming that you have another function for y,

% though you may use the same function as x. In this case,

% if y\_num is smaller than 96, it will be wrapped.

% E.g., y\_num is 10, then function value of 33 will be

% postion 3 for y frame. See below for implementation.

Pattern\_id = 3; % some random value as an example.

init\_pos = [3 2]; % The actual position of x, y frame is

% offset by these numbers.

Panel\_com('set\_funcx\_freq' , Freq);

Panel\_com('set\_posfunc\_id', [1, xfid]);

Panel\_com('set\_funcy\_freq' , Freq);

Panel\_com('set\_posfunc\_id', [2, yfid]);

Panel\_com('set\_mode', [4 4]); % both x and y are using mode 4 here,

% but you don’t have to.

Panel\_com('send\_gain\_bias', [0 0 0 0]);

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

Panel\_com('set\_position', init\_pos); % offset the position functions.

% Caution: Panel\_com automatically subtract 1 from init\_pos.

Panel\_com('start')

This code shows how the current index for the x channel, index\_x, is determined:

temp\_x\_val = (X\_pos\_index + function\_X[function\_counter]);

if (temp\_x\_val >= 0) {index\_x = temp\_x\_val%x\_num; }

if (temp\_x\_val < 0) {index\_x = x\_num - ((abs(temp\_x\_val))%x\_num); }

As you may notice, this implementation uses the function generator as an offset of the value stored in X\_pos\_index, which is set when a new X,Y position is sent to the controller. In this way the same function can be replayed, at different positions (in functions space), without recreating the function. Also note that the ‘%’ operation in C-code causes wrapping if the function value is larger than x\_num.

**mode 5, function generator debug mode** – in this mode, the controller does not update the display, it plays out the function generator through the DAC outputs. The 0-5V DAC output range is used to represent a -5 to + 5 range, so the zero of the function is shifted up to 2.5 V and the range is compressed by a factor of 2 (e.g. -5V is sent out as a zero, 0V as 2.5 V, + 5V as 5V).

**Technical Appendix 3 – System setup guide**

The components required to effectively perform an experiment in a visual flight arena are the controller, wingbeat analyzer (WBA), data acquisition module, oscilloscope, PC, and display panels. Directions and software for many of these components are available at <https://bitbucket.org/mreiser/panels/wiki/Home>

This document outlines the proper configuration of these components for a typical open- or closed-loop flight experiment. Once this setup works, you may change the configuration depending on your particular experiments.

**Basic Components:**

1. The flight arena consists of a circular array of 8x8 dot matrix displays of LEDs with additional electronics to drive the display.

2. The data acquisition module is the nidaq, or National Instruments Data Acquisition Device. In this case, the nidaq is a NI USB-6229 BNC Multifunction DAQ. This model has 16 analog inputs, 4 analog outputs, 8 DIOs and 2 user defined BNC terminals—more information about this device is available at <http://sine.ni.com/nips/cds/view/p/lang/en/nid/203866>.

3. The controller is a microprocessor circuit custom designed to operate the flight arena. The controller communicates with a PC through a serial port and to the panels using a rapid serial interface.

4. The wingbeat analyzer (WBA) measures the frequency and amplitude of the fly wing stroke. It also computes the sum and difference of the amplitudes of the left and right wings. This device is custom built by JFI Electronics at the University of Chicago. You may purchase the photo detector from the same vendor or assemble by yourself. You may also use any kind of IR LED for illumination. A typical IR LED is Emitter IR 880NM 8DEG TO-46 - PDI-E803 (PDI-E803-ND from Digi-key). Finally, a use of proper IR long pass filter is essential to avoid ambient light noise. Typically, RG-780 Long Pass Filter 1 inch Dia (NT32-757) is used.

5. The PC is used to display patterns on the flight arena and record data through the nidaq. The PC must be capable of running Matlab as well as a data acquisition program such as Spikehound (freely available), Matlab data acquisition toolbox or AxoScope. It must have at least two USB and one serial ports (one USB and one serial ports for older controllers).

6. An oscilloscope with three input channels and an external trigger will suffice.

**Connectivity:**

|  |  |  |  |
| --- | --- | --- | --- |
| Controller | ADC0 | WBA L |  |
| Controller | ADC1 | WBA R |  |
| Controller | DAC0 | Nidaq Input 3 |  |
| Controller | DAC1 | Nidaq Input 4 |  |
| Controller | PC1 USB | PC USB |  |
| Controller | I2C | Serial Port to Arena |  |
| Nidaq | Analog Input 0 | WBA - L |  |
| Nidaq | Analog Input 1 | WBA - R |  |
| Nidaq | Analog Input 2 | WBA - Frequency |  |
| Nidaq | Analog Input 3 | Controller DAC0 |  |
| Nidaq | Analog Input 4 | Controller DAC1 |  |
| Nidaq | Analog Input 5 | WBA - L-R |  |
| Nidaq | Analog Input 6 | Analog Output 1 |  |
| Nidaq | USB Output | PC |  |
| Nidaq | Analog Output 1 | Analog Input 6 |  |
| Wingbeat Analyzer | WBA L | Controller ADC0, Nidaq AI 0 |  |
| Wingbeat Analyzer | WBA R | Controller ADC1, Nidaq AI 1 |  |
| Wingbeat Analyzer | Signal Out L | Oscilloscope Ch 1, Speakers |  |
| Wingbeat Analyzer | Signal Out R | Oscilloscope Ch 2, Speakers |  |
| Wingbeat Analyzer | WBA L-R | Oscilloscope Ch 3, Nidaq AI 5 |  |
| Wingbeat Analyzer | SYNC (-TTL) | Oscilloscope Ext Trig |  |
| Wingbeat Analyzer | LED Out | IR LED on Flight Arena |  |
| Wingbeat Analyzer | Photo detector | Photo detector on Flight Arena |  |
| Wingbeat Analyzer | Frequency | Nidaq AI 2 |  |
| Oscilloscope | Channel 1 | WBA L |  |
| Oscilloscope | Channel 2 | WBA R |  |
| Oscilloscope | Channel 3 | WBA L-R |  |
| Oscilloscope | Ext Trig | WBA SYNC (-TTL) |  |
| Flight Arena | Serial Port | Controller I2C |  |
| Flight Arena | Photodetector | WBA Photodetector |  |
| Flight Arena | LED | WBA LED Out |  |
| PC | USB | Nidaq |  |
| PC | USB | Controller PC1 USB |  |

Note that ADC0~3 corresponds to ADC1~4 for older controllers. Similarly, DAC0~3 to DAC1~4 and INT0~3 to INT1~4.

If you want to use mode 3 in the new controller, you should make a custom connector to the parallel port on the back of the controller, because ADC4 and 5 are not readily available in the front panel of the controller. See <https://bitbucket.org/mreiser/panels-hardware/wiki/Controller> for more details.