NIMH MonkeyLogic Manual

Jaewon Hwang, Ph.D.

Staff Scientist

Section on Neurobiology of Learning and Memory (SNLM)
Laboratory of Neuropsychology (LN)
National Institute of Mental Health (NIMH)
National Institutes of Health (NIH)

Contents

1. About NIMH MonkeyLogic

NIMH MonkeyLogic (NIMH ML) is a MATLAB-based software tool for behavioral control and data acquisition. It allows users to design sensory, motor, or cognitive tasks with a familiar, high-level language (i.e., MATLAB) and execute them with high temporal accuracy. Many object types are available for task composition, such as image, movie, sound, TTL and analog output, and data can be collected during tasks from various signal sources (analog and digital input, mouse/touchscreen, USB joystick, TCP/IP eye tracker, etc.). The timing of events can be synchronized with external devices (e.g., Plexon, TDT, Blackrock) via event code exchanges.

NIMH ML started with **NIMH DAQ Toolbox** and **MonkeyLogic Graphics Library (MGL)**. NIMH DAQ Toolbox was developed to extend the legacy DAQ interface of MATLAB to the 64-bit environment and support real-time behavior monitoring with one data acquisition device. MGL was written to provide an information-rich replica of the subject screen, as well as support for transparent images, movie streaming and low-latency audio output. At first NIMH DAQ Toolbox and MGL were used to power the original MonkeyLogic on the latest computing environment and released with a modified version of the original MonkeyLogic. That made the 1st version of NIMH ML.

The 2nd version of NIMH ML is completely re-written from scratch in the object-oriented programming style and uses **its own data file format (BHV2)**. It is also equipped with upgraded NIMH DAQ Toolbox and MGL, which support more input devices and more features, and provides a new scripting framework that can handle more complex graphics and behavior.

2. Features

- Full support for latest 64-bit MATLAB
- Compatible with the original MonkeyLogic's behavioral tasks
- NIMH DAQ Toolbox
    - Real-time behavioral monitoring (1-ms resolution) using only one DAQ board
    - Support for more input devices, including mouse/touchscreen, USB joystick and TCP/IP eye tracker
- MonkeyLogic Graphics Library (MGL)
    - "What you see is what your monkey sees."
    - Support for transparent images by alpha blending or color key
    - Movie streaming (no limit to movie length)
    - Low-latency audio output with XAudio2
- mlplayer: a trial-replay and video-exporting tool
- Simulation mode that allows testing user tasks with no special hardware

3. System Requirements

- Windows 7 or later
- MATLAB R2011a or later
    - No MATLAB toolbox is required.
- Microsoft Visual C++ 2013 Redistributable[1]
    - Download from https://www.microsoft.com/en-us/download/details.aspx?id=40784
- DirectX End-User Runtimes[1]
    - Download from https://www.microsoft.com/en-us/download/details.aspx?id=8109
- National Instruments Multifunction I/O Device (optional)
    - No need to install two boards (unless many connections are necessary)
    - USB-type devices are supported.[2]

1. NIMH ML will open the webpages upon staring up, if those libraries are not installed on the system.

2. The sample transfer rate of USB devices may depend on the system. It is recommended to test the actual transfer rate on the machine that will be used. Run the "\task\benchmark\2 sample transfer" task.

4. Getting Started

Obtaining NIMH MonkeyLogic

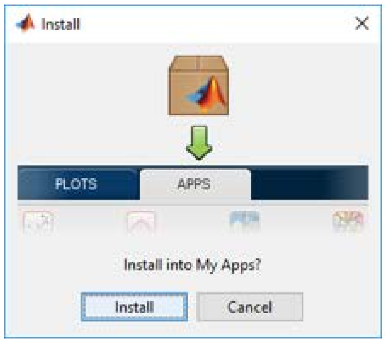
NIMH ML can be obtained from https://goo.gl/wuxWg7

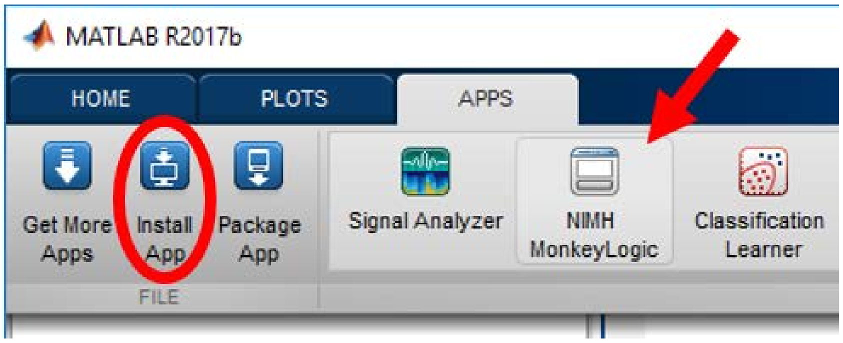
Software Installation

You can use either a MATLAB app installer (R2012a or later) or a zip file.

Using a MATLAB app installer

Double-click the downloaded *.mlappinstall file. It will open MATLAB and pop up a question dialog as below. Click the [Install] button and NIMH ML will be added to the MATLAB menu. If this process fails for any reason, you can manually open MATLAB and install the package by clicking the [Install App] menu.





The installation directory is dependent on the version of your MATLAB.

\your user directory\Documents\MATLAB\Add-Ons\Apps (R2015b or later)
\your user directory\Documents\MATLAB\Apps (R2015a or earlier)

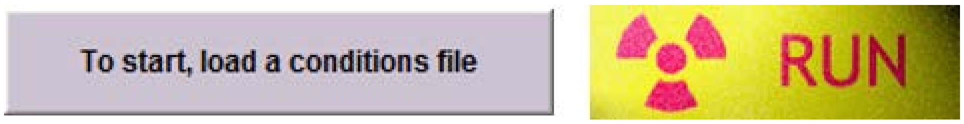
Using a zip file

Decompress the zip file to a directory that you choose and add the directory to the MATLAB path. You can add the subdirectories as well, but it is not necessary.

Starting NIMH MonkeyLogic

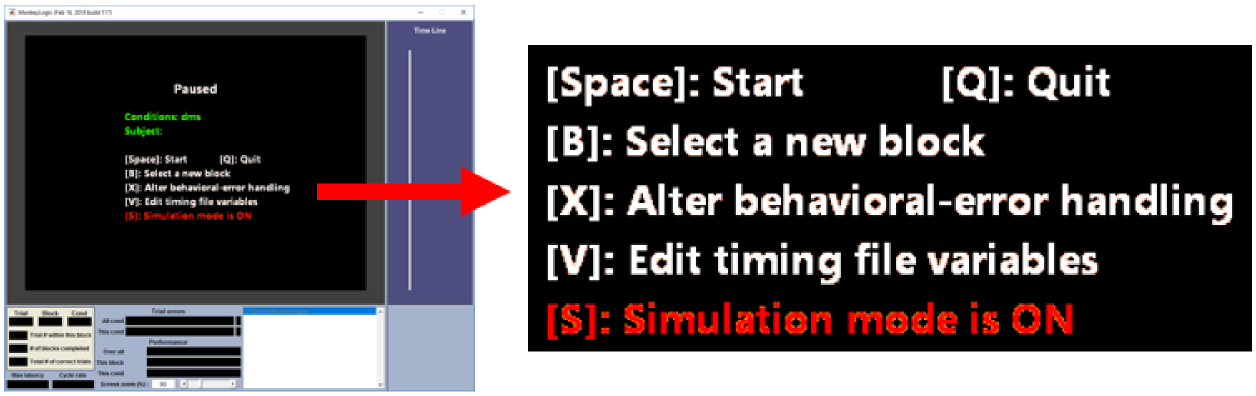
Click the [NIMH MonkeyLogic] icon on the MATLAB APPS menu (if you installed with the MATLAB app installer) or type "monkeylogic" on the MATLAB command window (if you installed with the zip file), depending on your installation method.

NIMH ML comes with a delayed match-to-sample task and many other examples. They are under the "task" directory of the ML directory. To start a task, choose a conditions file by clicking the [Load a conditions file] button on the ML GUI (left in the figure below) and then hit the [Run] button.



You can run a task without any DAQ board or input device, by activating the **simulation mode** in the pause menu. In the simulation mode, most of common input signals are replaced with mouse and key inputs, like the following.

- Eye: mouse
- Joystick: cursor keys
- Touch: mouse click
- Buttons: key '1' to key '0'



Paused

Conditions: dms
Subject:

[Space]: Start [Q]: Quit
[B]: Select a new block
[X]: Alter behavioral-error handling
[V]: Edit timing file variables
[S]: Simulation mode is ON

Data Files Supported by NIMH MonkeyLogic

NIMH ML supports its own data file format, called BHV2 (*.bhv2), as well as HDF5 (*.h5) and MAT (*.mat). **BHV2** is a private format that is based on a simple recursive algorithm (see "**BHV2 Binary Structure**" in the appendix). It provides decent read and write performance and is most recommended. **HDF5** is supported by many commercial and non-commercial software platforms, including Java, MATLAB, Scilab, Octave, Mathematica, IDL, Python, R and Julia, but its read performance in MATLAB is a bit disappointing. **MAT** is MATLAB's native data format and NIMH ML uses MAT-file Version 7.3. MAT has a problem that it gets slower as more and more variables are stored, even though the file compression is disabled.

The **mlread** function provides a unified read interface for all the formats. It returns trial-by-trial data in a 1-by-n array of structures.

```
data = mlread;
data = mlread(filename);
[data, MLConfig, TrialRecord, filename] = mlread(__);
```

The **mlconcatenate** function combines trial-by-trial analog data into one large seamless matrix and adjusts all timestamps accordingly, as if they are recorded in one single trial. This function is useful when reading data files in which signals were continuously recorded through inter-trial intervals.

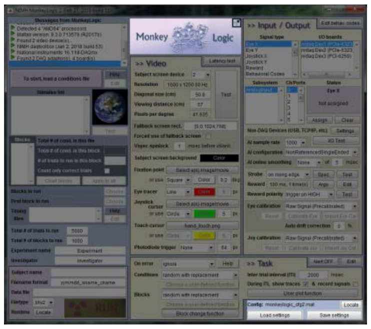
```
data = mlconcatenate;
data = mlconcatenate(filename);
[data, MLConfig, TrialRecord, filename] = mlconcatenate(__);
```

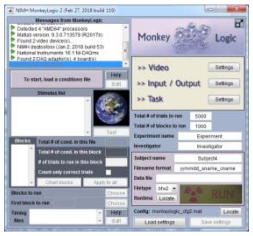
If you saved the file-source stimuli in the data file, you can retrieve them with the **mlexportstim** function.

```
mlexportstim;
mlexportstim(destination_path, datafile);
```

5. Main Menu

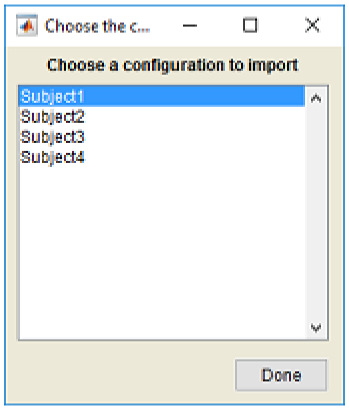
Logo & Load/Save Settings





[Collapsed menu]

- **Three monkeys logo:** Opens the NIMH ML homepage (https://goo.gl/wuxWg7)
- **Collapse/Expand menu button (at the top-right corner of the logo):** Hides/shows the sub-menus (Video, Input/Output and Task). The collapsed menu is useful when running NIMH ML on a small screen.
- **Config [Locate] button:** Opens the config file folder. It is usually the current task directory.
- **Load settings:** Once a config file is chosen, a pop-up window will show up as below and ask which subject's configuration you want to load from the file. <u>Note that **"# of trials to run in this block"**, "**Count correct trials only**", "**Blocks to run**", "**First block to run**" and "**Subject name**" are not overwritten, if the settings are loaded in this way</u>. See "**Subject name**" in the "**Conditions File & Run Button**" section for more information.



Choose the c...

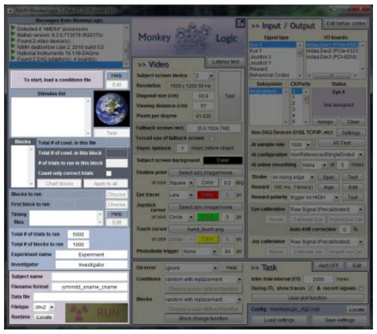
Choose a configuration to import

Subject1
Subject2
Subject3
Subject4

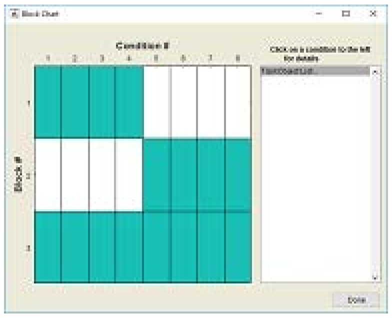
Done

- **Save settings:** Stores the current settings to the config file (*_cfg2.mat). The button is activated only when there is something changed. If any change is about to be overwritten without being saved, a file save dialog will pop up. The current config will be saved without asking, when the 'RUN' button is clicked.

Conditions File & Run Button



- **[Load conditions file] button (“To start, load …”):** Opens a conditions/userloop file.
    - **[Help] button:** Opens the conditions file manual.
    - **[Edit] button:** Opens the current conditions file in the text editor. The conditions file should be reloaded after edits have been completed.
- **[Stimulus list] pane:** Shows the list of the stimuli found in the conditions file. <u>In case that the opened file is a userloop function, it will display ‘user-defined’</u>.
    - **Earth icon:** Displays the stimulus selected in the [stimulus list] pane.
    - **[Test] button:** Displays/plays the selected stimulus, as if it is displayed/played during a trial. Pressing any key will stop the test.
- **Total # of cond. in this file:** Displays the total number of conditions found in the selected conditions file.
- **[Blocks] pane:** Displays the available blocks.
    - **Total # of cond. in this block:** Displays the total number of conditions associated with the selected block.
    - **# of trials to run in this block:** Edit this value to determine the number of trials to be run in the selected block.
    - **Count correct trials only:** As it states.
    - **[Chart blocks] button:** Opens a figure that shows which conditions appear in which blocks, such as the following one.



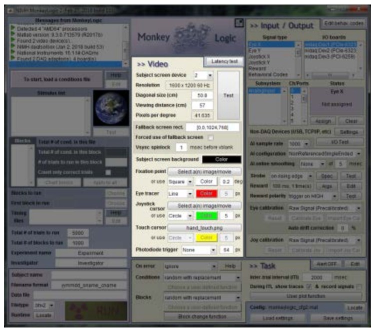
    - **[Apply to all] button:** Applies the changes to all blocks.

- **Blocks to run:** Only the blocks selected here will be used during task execution. Initially all blocks are selected.
- **First block to run:** Select the first block to run, when the task starts. 'TBD (to be determined)' will let ML determine it, according to the block selection logic in the Task submenu.
- **Timing files:** Lists the timing scripts used by the current conditions file. In case that a userloop function is loaded, 'user-defined' will be displayed instead. Double-clicking the timing script name will open its runtime file (or the runtime folder if the runtime file is not created yet). The runtime file is a custom MATLAB function that is created by combining the user timing script and a runtime function provider (trialholder_v1.m or trialholder_v2.m) when the RUN button is hit. What ML runs to start the task is this runtime file, not the timing script itself.
  - **[Help] button:** Opens the timing script manual.
  - **[Edit] button:** Opens the selected timing file in the MATLAB editor.
- **Total # of trials to run:** Determines when the task will stop if not manually terminated. This includes both correct and incorrect trials.
- **Total # of blocks to run:** Determines the total number of blocks (not block numbers) to be run; the task will end once this number of blocks has executed (or the "**Total # of trials to run**" value has been reached, whichever comes first).

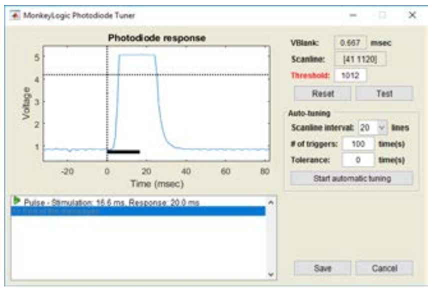
- **Subject name:** NIMH ML keeps a separate configuration profile per each subject. If the entered subject name is new, then the current settings are copied under the new name and the "Save settings" button is activated. If the entered name already exists in the configuration file, the settings previously saved under the name are loaded automatically.
  - Unlike the original ML, NIMH ML keeps the last changes of editable variables in the configuration file (per each subject) and does not read the variables from the timing script ever again. When a new subject's profile is created, the values of editable variables are also copied from the current subject's configuration. Therefore, in NIMH ML, editable variables are never reset to the initial values written in the timing script, once read. So, to modify their values, do not edit timing scripts. Use the editable variable dialog in the "**Pause menu**".
  - If there are unsaved changes when a new name is typed, a pop-up window will ask if you want to save the changes. However, it is not for the new name you just typed, but for the name there before you typed. Therefore, the "Save settings" button will stay active, even after you answer "Yes" to the pop-up.
  - When the task is loaded next time, the last subject's configuration will be loaded automatically.

- **Filename format:** Set the format of the default data file name.
    - **‘expname’ or ‘ename’:** Experiment Name
    - **‘yourname’ or ‘yname’:** Investigator
    - **‘condname’ or ‘cname’:** Conditions file name
    - **‘subjname’ or ‘sname’:** Subject name
    - **yyyy:** Year in full (1990, 2002)
    - **yy:** Year in two digits (90, 02)
    - **mmm:** Month using first three letters (Mar, Dec)
    - **mm:** Month in two digits (03, 12)
    - **ddd:** Day using first three letters (Mon, Tue)
    - **dd:** Day in two digits (05, 20)
    - **HH:** Hour in two digits (05, 24)
    - **MM:** Minute in two digits (12, 02)
    - **SS:** Second in two digits (07, 59)
- **Data file:** Leave this field blank if you want it to be filled with the formatted name.
- **Filetype:** See “**Data Files supported by NIMH MonkeyLogic**” in the “Getting Started” chapter.
- **Save stimuli:** When this option is checked, stimuli used during the task are saved in the data file. All stimuli created from file source will be saved. Saved stimuli can be extracted by using the **mlexportstim** function.
- **[RUN] button:** Activated when a task is loaded. Upon a click, the current configuration is automatically saved, and the task begins.

Video

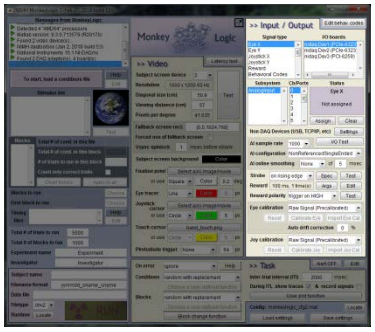


- **[Latency test] button:** Runs a benchmark trial.
- **Subject screen device:** Lists the screens available on the system.
- **[Test] button:** Displays an animation on the selected screen, to identify it.
- **Resolution:** Pixel width, height & refresh rate of the selected screen. To change the resolution, use [Windows Display Settings].
- **Diagonal size (cm):** Enter the diagonal screen size (in centimeters) of the subject's screen.
- **Viewing distance (cm):** Enter the distance (in centimeters) from the subject's eye to the screen.
- **Pixels per degree:** This number is used by ML to compute the degrees of visual angle. It allows users to specify stimulus coordinates in degrees instead of pixels.
- **Fallback screen rect.:** This fallback screen is a windowed subject screen and is used for testing when there is only one monitor. Set where you want to see it on the screen. [LEFT TOP RIGHT BOTTOM] in Windows' pixel coordinates.
- **Forced use of fallback screen:** Forces the subject screen to be displayed in a window.
- **Vsync spinlock:** This is a period before the vertical blank time in which ML stops other jobs and just waits for screen flips occurring. Applicable to the runtime version 1 only.
- **Subject screen background:** Background color of the subject screen.
- **Fixation point, Eye tracer, Joystick cursor, Touch cursor:** Allows you to use either an image/movie or a circle/square of given color and size. Eye tracer and Touch cursor are shown only on the control screen.
- **Photodiode trigger:** Displays black and white squares in turns at the selected location to drive the photodiode.
    - **[Tune] button:** Launches the photodiode tuning tool. This tool adjusts the timing of screen flip commands based on the response characteristics of the monitor and the photodiode so that stimuli can be presented close to the time of eventmarkers. A threshold (scanline #) can be set manually or automatically. Threshold 0 means that the screen will be flipped during the vertical blank time.



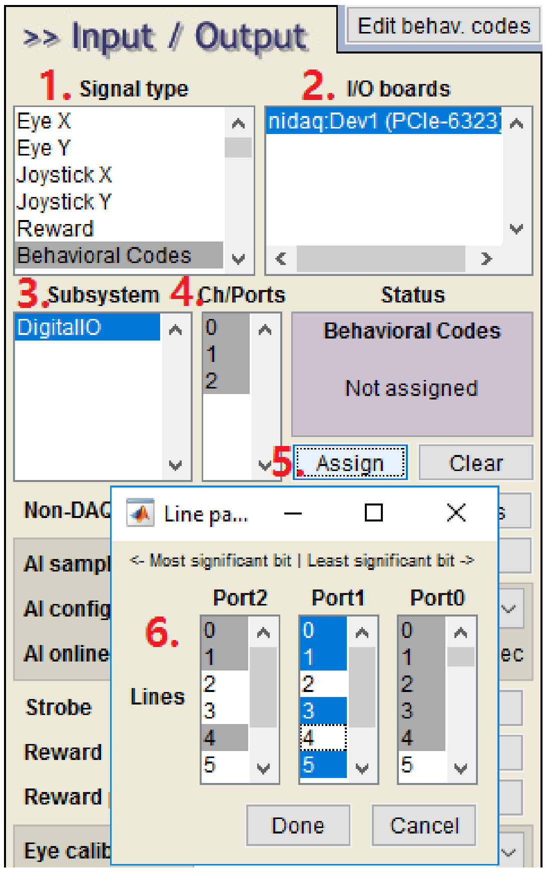
VBlank: 0.667 msec
Scanline: [41 1120]
Threshold: 1012
Auto-tuning
Scanline interval: 20 lines
# of triggers: 100 time(s)
Tolerance: 0 time(s)
Pulse - Stimulation: 16.6 ms, Response: 20.0 ms

Input / Output



- **[Edit behav. codes] button:** Opens the “codes.txt” file that lists Behavioral Code numbers of the current task with their associated descriptions. If codes.txt exists both in the ML directory and the current task directory, the one in the task directory is opened and used for the task. You can also define this code-description association in the timing script with the **bhv_code** command.

DAQ board settings



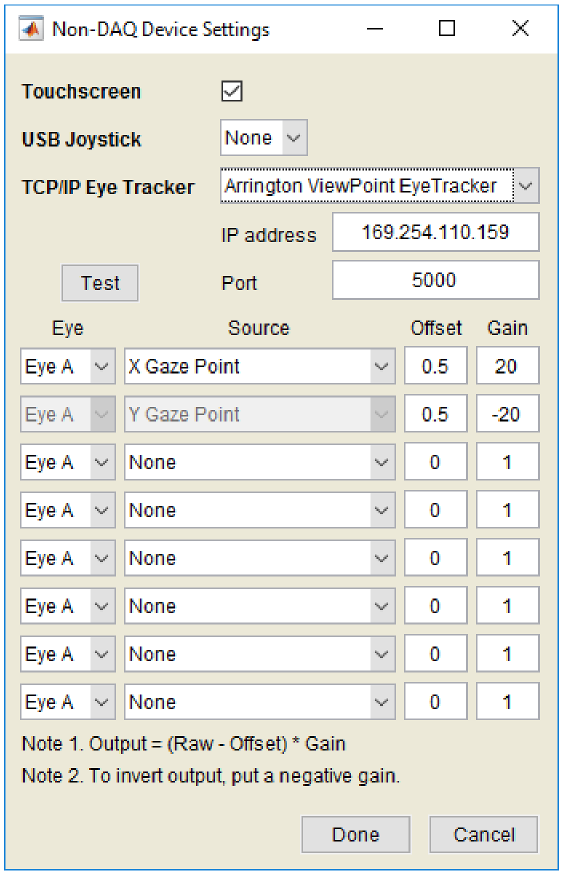
- Click the panels in the order shown on the left, to map behavioral signals with DAQ channels/ports that they are connected to. If you do not have this connection information, ask someone who knows it or visually check the wire connection with the device pinout diagram (see **Device Pinouts**).

- Eye X &Y (and Joystick X & Y) must be assigned on the same board together. Otherwise, an error message will be displayed, when starting tasks.

- Multiple channels/ports can be assigned to Reward and Behavioral Codes. Drag on the panel or use SHIFT + CLICK and CTRL + CLICK combinations.

- In case of digital lines, a line selection dialog will appear additionally, on clicking the [Assign] button.

Non-DAQ devices (USB, TCP/IP, etc.)


Non-DAQ Device Settings

Touchscreen ☑

USB Joystick: None

TCP/IP Eye Tracker: Arrington ViewPoint EyeTracker

IP address: 169.254.110.159

Port: 5000

Eye	Source	Offset	Gain
Eye A	X Gaze Point	0.5	20
Eye A	Y Gaze Point	0.5	-20
Eye A	None	0	1
Eye A	None	0	1
Eye A	None	0	1
Eye A	None	0	1
Eye A	None	0	1
Eye A	None	0	1

Note 1. Output = (Raw - Offset) * Gain

Note 2. To invert output, put a negative gain.

- These devices do not have an ADC (analog digital converter), but NIMH ML can monitor their output every millisecond based on the software timer.

- **Touchscreen**
  - This option should be checked to store mouse/touch tracking data to the data file. The tracking data can be retrieved from AnalogData.Mouse and its format is [X Y LeftButton RightButton].

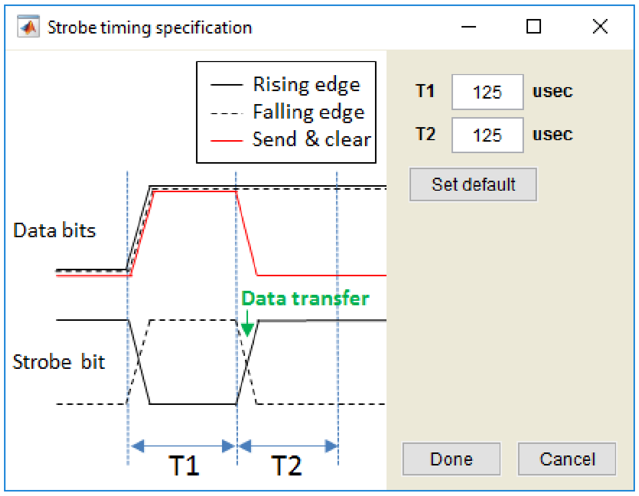
- If a USB joystick or TCP/IP eye tracker is selected here, it has priority over the one connected to the DAQ board.

- **TCP/IP Eye Tracker**
  - Currently Arrington Research's ViewPoint EyeTrackers and SR Research's EyeLink trackers are supported.
  - **[Test] button:** Test the TCP/IP connection.
  - The first two signal sources must be X & Y gaze points (displacement signals). Some GUI components may be disabled, to make it sure.
  - Users need to disable Nagle's algorithm as instructed below, to get the maximum TCP/IP performance. Nagle's algorithm is turned on, by default, in Windows.

1. To get into the registry editor, go to Start > Run > Type: regedit
2. Browse to: **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces**
3. Find the network interface that you want to modify. You can search for the IP address assigned to the network adapter.
4. Create two new entries for the interface. You can do this by right-clicking the interface in the left pane and going to New > DWORD (32-bit) Value.
   - Name the first one "**TcpAckFrequency**" (case sensitive).
   - Name the second one "**TCPNoDelay**" (again, case sensitive).
5. Double-click each of your new entries and set their 'Value Data' to "**1**". Leave "Hexadecimal" checked under 'Base'.
6. Exit your registry and reboot.

(Continued to the **Input / Output** menu)

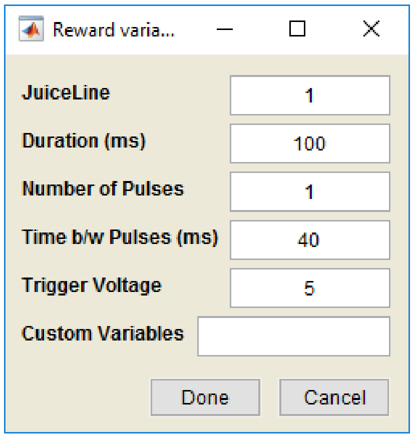
- **[I/O Test] button:** Starts a task that can test the assigned I/O.
- **AI sample rate:** NIMH ML always acquires samples at 1 kHz, regardless of this setting. Then it skips some samples at the time of data saving, if this number is set below 1000. In this way, NIMH ML can monitor behavior with high resolution but keep the data size small.
- **AI configuration:** Sets the AI ground configuration of the DAQ board. See "**Analog Input Ground Configuration**" in the "NI Multifunction I/O Device" chapter.
- **AI online smoothing:** Smooths input signals to remove noise artifacts. Applicable to the runtime version 1 only.
- **Strobe:** Set whether the Behavioral Codes (event markers) will be sent on the rising-edge or falling-edge of the Strobe Bit. There is also a third option that does not require the Strobe Bit ("Send and Clear") for systems that capture the codes based on change detection.
    - o **[Spec] button:** You can change the duration of Strobe Bit pulses. See the figure below.
    - o **[Test] button:** Sends a test Strobe Bit.



T1 125 usec

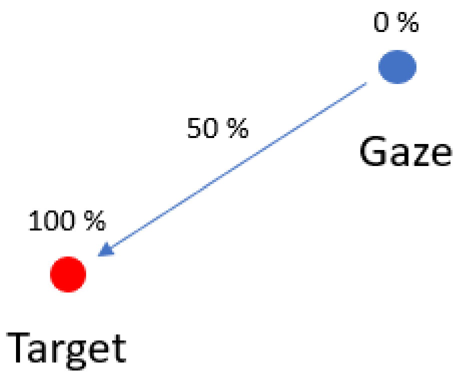
T2 125 usec

- **Reward**
  - **[Args] button:** Edit the reward function arguments (i.e., goodmonkey) for testing.
  - **[Edit] button:** Opens the current reward function. By default, it opens reward_function.m in the ML directory. If there is a file in the current task directory with the same name, the one in the task directory has a priority.

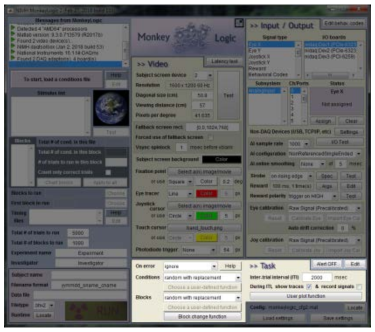

Reward varia...

JuiceLine	1
Duration (ms)	100
Number of Pulses	1
Time b/w Pulses (ms)	40
Trigger Voltage	5
Custom Variables	

- **Reward polarity:** Sets whether the reward device will be triggered on a TTL HIGH or LOW signal.
  - **[Test] button:** Sends out a test reward pulse.
- **Eye calibration & Joy calibration:** See the "**Calibrating Eye/Joystick Signals**" chapter.
  - **[Reset] button:** Clears the calibration matrix of the currently selected method.
  - **[Calibrate / Re-calibrate] button:** Opens the selected calibration tool.
  - **[Import] calibration button:** Copies the calibration matrix of the currently selected method from another configuration file.
- **Eye [Auto drift correction]:** If this percentage is larger than 0 (the maximum is 100), the positions of the fixation target and the gaze are compared at the end of each trial. If there is a difference, the calibration matrix is updated to translate the eye position toward the fixation target according to the number given here. Putting zero disables the auto correction.



Task



- **[Alert] button:** Turns on/off the alert state. If it is ON, NIMH ML calls alert_function.m when special events occur, such as task_start, block_start, trial_start, etc. See alert_function.m in the ML directory for details. You can catch those hooks in the function and make ML do special things, such as sending you a text message and starting an external device.
    - **[Edit] button:** Opens the alert_function.m in the ML directory. If you have a copy of this file in the task directory, the one in the task directory is used instead.
- **On error:** What to do if the subject makes an error.
    - **Ignore** will simply disregard errors when the next condition is selected.
    - **Repeat immediately** will cause the same condition to be played repeatedly until the correct response is made (i.e., trialerror(0) has been set by the timing file).
    - **Repeat delayed** will throw that condition into the queue, but at a later point in the block; this can happen an indefinite number of times for a given condition. If a particular condition is repeatedly performed incorrectly, increasing numbers of copies of this condition will "pile-up" later in the block, though still interspersed with whatever other conditions still remain to be played.
- **Conditions:** Determines the method used to select the next condition.
    - **Random with replacement** causes the next trial's condition to be selected randomly, without regard to which conditions have already been used.
    - **Random without replacement** will randomly choose the next trial's condition but will exclude those trials already used.
    - **Increasing and Decreasing** simply choose the next condition in numerical order, rising or falling as appropriate. The condition numbers will "wrap" when the end or beginning of the condition list has been reached.
    - **User-defined** allows the writing of a MATLAB function ("**condition-selection function**") to determine how conditions are selected. This is useful to enact particular rules about how certain conditions should follow others, or how the selection of certain conditions preclude others, for example. This function should be contained in a typical m-file, should expect the TrialRecord structure as input and should return a single scalar, the condition to-be-run.

- **Blocks:** Identical in logic to Conditions above. The **User-defined** option here requires a MATLAB function that takes the TrialRecord structure as input and returns a new block number ("**block-selection function**"). Note that the function will be run on each trial to determine if a new block should be chosen. It is useful, for example, when behavior must reach a certain threshold before allowing the next block to proceed.
- **Block-change function:** This user-created function, if specified, is called after every trial to determine whether a new block should be initiated. It is passed the TrialRecord structure, and should return a value of "1" to change blocks, or "0" to continue the current block. The next block will be selected in the usual manner, according to the options described above. However**,** if the Block-Selection Function (above) is set to the same file as the Block-Change Function, this indicates that the Block-Change Function returns not simply a block-switch flag, but actually determines the next block to be run (e.g., return "3" to switch to running the third block). This feature allows one m-file to control both when a block switches and which block is selected next.
- **Inter-trial interval (ITI):** Enter the desired inter-trial interval (in milliseconds).
- **During ITI,**
  - **Show traces:** Determines whether to display used stimuli and traces of input signals on the control screen during the ITI.
  - **Record signals:** Determines whether to record input signals during the ITI. If this option is selected, the ITI start time is regarded as the beginning of the next trial.
- **User plot function:** Name of a MATLAB function that will draw to the space normally occupied by the reaction time histogram on the control screen. Users can use this function to display results of online trial-by-trial analysis. The function is supposed to take the TrialRecord structure as input and return nothing.

6. Creating a Task

The best way to learn how to write your own task is to begin with an existing example and try modifying it. There are many example tasks under the "task" folder of the ML installation directory.

Conditions File & Timing Script (Runtime Library Version 1)

NIMH ML supports the original ML's behavioral tasks, so a new task can be written in the traditional style using a conditions file and a timing script. The conditions file is a tab-delimited text file which lists a set of stimuli that can occur in each trial. Each stimulus is referred to as a "TaskObject" and can take the form of a visual object, a sound, or an analog or TTL output. The timing script is a MATLAB program that determines when and under what conditions each of those stimuli is presented. For the structure of these files and how to write them, please refer to **Structure of Conditions File** & **Runtime Library Functions for Timing script** in the appendix.

In NIMH ML, the userloop function and the runtime library version 2 can be substituted for the conditions file and the runtime library version 1, respectively.

Userloop Function

The conditions file requires defining all trial conditions explicitly. This may not be convenient sometimes, for example, when there are so many conditions and stimuli or when the task needs a flexible way of randomizing trial sequences and handling error trials. <u>A userloop function is a MATLAB function that feeds the information necessary to run the next trial, in lieu of the conditions file</u>. It is called before each trial starts and allows user to determine which TaskObject and which timing script will be used for the trial on the fly. You can also preload large stimuli in the userloop function and reuse them so that the inter-trial interval does not get increased by the stimulus creation time. The details of the userloop function are well-documented in the example tasks that are under the "task\userloop\1 dms with userloop" directory of the ML installation folder. See the dms_userloop.m file.

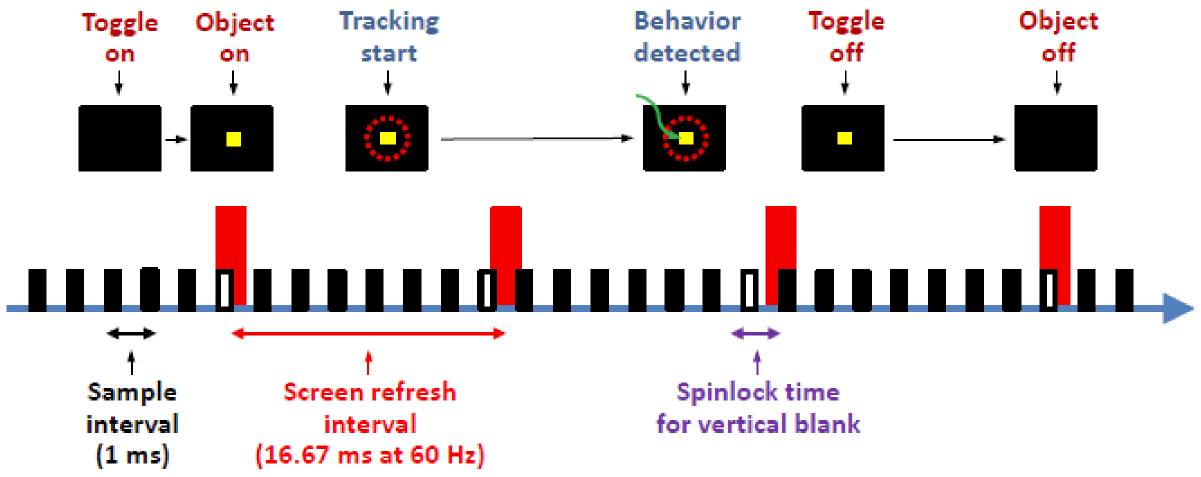
Runtime Library Version 2 (“Scene Framework”)

Background

In the runtime library v1, stimulus presentation and behavior tracking are handled by `toggleobject()` and `eyejoytrack()`, respectively. This framework is a little disadvantageous in designing dynamic, interactive stimuli, due to the following reasons.

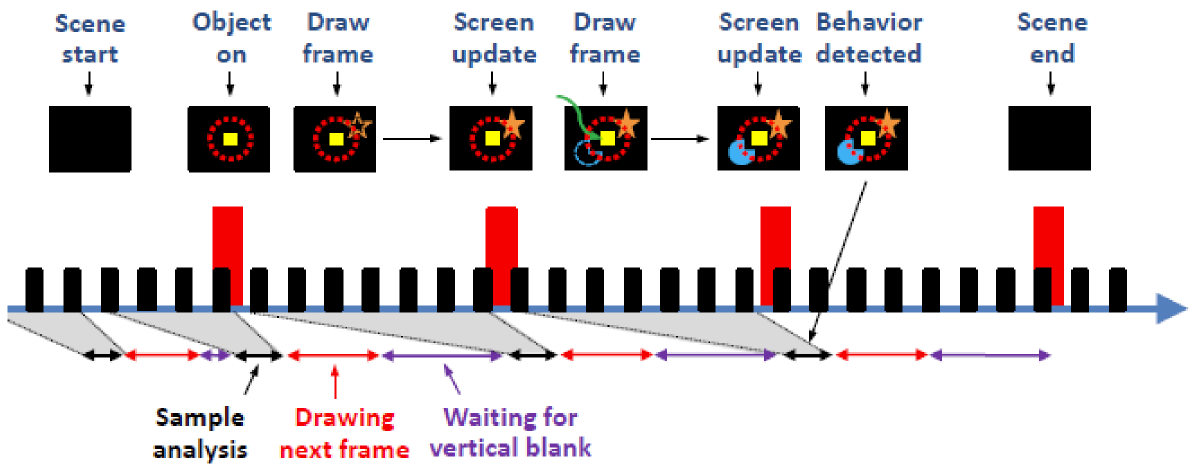
1. Stimuli and behavior are processed separately and there is no proper way to change stimuli during behavior tracking.
2. While tracking behavior, `eyejoytrack()` tries to read a new sample at 1-ms intervals or faster, which leaves us too short time to perform sophisticated computation or draw complex stimuli.
3. Because `toggleobject()` and `eyejoytrack()` have many optional arguments, the cost of switching between two functions is high.

```
toggleobject(taskobject, ‘eventmarker’, eventcode);
eyejoytrack(‘acquirefix’, taskobject, threshold, duration);
```



The runtime v2 takes a different approach. In this new runtime, behavior tracking and stimulus presentation are both handled by one function, `run_scene()`. In addition, samples collected during one refresh interval are analyzed all together at the beginning of the next refresh interval and the screen is redrawn based on the sample analysis.

```
scene = create_scene(adapter, taskobject);
run_scene(scene, eventcode);
```



Therefore, the cycle of [analyzing samples]-[drawing screen]-[presenting] is repeated each frame and, by tapping into this cycle, we can see what happened in behavior and then decide what to show on the screen.

One disadvantage of this approach is that we don't know when the behavior occurred until the next frame begins. (See the time of the behavior occurrence, the green arrow, and the time of behavior detection in the above figure.) However, this cannot be a big issue for the following reasons.

1. We cannot update the screen contents until the next vertical blank time anyway, so it is not always necessary to detect behavior immediately. (If you use audio stimuli only, that is a different story and you can stay with `toggleobject()` and `eyejoytrack()` in that case.)

2. We may detect behavior a little later (by one refresh cycle at most), but we don't lose any information. We can still get the exact time when the behavior occurred. What is not possible is to call `eventmarker()` to stamp the reaction time as soon as the behavior occurs. However, the window-crossing time cannot be an accurate measure of the reaction time, considering the size of the fixation window is arbitrary. If you are serious about reaction times, you probably want to use a velocity criterion, which requires some offline analysis.

In spite of some limitations, this approach has advantages in dynamic and precise frame-by-frame control of visual stimuli. In fact, it is the way how most of game software handles graphics.

Scene Framework

In the runtime library v2, `toggleobject()` and `eyejoytrack()` are replaced by two new functions, `create_scene()` and `run_scene()`. `create_scene()` receives an "adapter" as the input argument and return a “scene” structure. Then, `run_scene()` renders a scene with it.

```
scene = create_scene(adapter [,taskobject]);
flip_time = run_scene(scene [,eventcode]);
```

The adapter is a MATLAB class object and a building block of a scene. You can make your own adapter or use the built-in ones. There are already ~40 adapters included in NIMH ML that cover almost everything you can do with the runtime v1 (and more). Please refer to the manual page included in the NIMH ML distribution package about the usage of the adapters. To make your own, make a copy of ext\ADAPTER_TEMPLATE.m and fill in the code.

As the function name indicates, `toggleobject()` of the runtime v1 turns on and off the stimulus object at each call.

```
 toggleobject(1);  % turn on Object #1
toggleobject(1);  % turn off Object #1
```

So, if you don't make the second call, the object stays on the screen. However, in the `create_scene()`, the meaning of the taskobject argument is "the objects that are needed to compose the scene", so they stay on the screen only while the scene is being presented. If you want to show the objects across multiple scenes, the object numbers should be provided to every `create_scene()` calls.

The return value of `create_scene()`, scene, becomes the input of `run_scene()`. The optional argument of `run_scene()`, eventcode, is the markers to stamp at the moment the stimuli are presented on the screen. And the return value, flip_time, is the time when it occurs.

Adapters

Multiple adapters can be concatenated as a chain, to detect complex behavior or draw complex stimuli. Below is an example (the file is under the task\runtime v2\0 green star). This example displays a green star on the center of the screen for 5 sec. To create this scene, three adapters are used.

```
----- Beginning of green_star.m -----
% create a chain of [NullTracker] -[TimeCounter]-[PolygonGraphic]
tc = TimeCounter(null_);
star = PolygonGraphic(tc);

% set the properties of the adapters
tc.Duration = 5000;         % in milliseconds
```

```
star.EdgeColor = [0 1 0];   % [r g b]
star.FaceColor = [0 1 0];
star.Size = 2;              % 2 deg by 2 deg
star.Position = [0 0];
star.Vertex = [0.5 1; 0.375 0.625; 0 0.625; 0.25 0.375; 0.125 0; ...
    0.5 0.25; 0.875 0; 0.75 0.375; 1 0.625; 0.625 0.625];

scene = create_scene(star); % call create_scene if the property setting is done

% run scene
run_scene(scene);
----- End of green_start.m -----
```

The first adapter is `NullTracker` (`null_`). All adapter chains must start with a special adapter called Tracker. There are 5 trackers and they are all pre-defined with reserved names: `eye_`, `joy_`, `touch_`, `button_` and `null_`. Each tracker reads new samples from the device that its name designates. `null_` does not read any data. The second adapter is `TimeCounter` that measures elapsed time. The third one is `PolygonGraphic` which draws a star in green.

This is what the TimeCounter adapter looks like.

```
----- Beginning of TimeCounter.m -----
classdef TimeCounter < handle
    properties  % user variables, readable & writable
        Duration = 0
    end
    properties (SetAccess = protected)  % read-only to users
        Success  % status variable that indicates whether Duration is passed
    end
    properties (Access = protected)  % internal variables, not accessible to users
        Adapter  % the underlying adapter, PolygonGraphic in this case
    end

    methods
        function obj = TimeCounter(varargin)  % constructor
            if 0==nargin, return, end
            obj.Adapter = varargin{1};  % store the underlying adapter, PolygonGraphic
        end
        function continue_ = analyze(obj,p)  % sample analysis
            obj.Adapter.analyze(p);          % call PolygonGraphic's analyze()
            obj.Success = obj.Duration <= p.scene_time();
            continue_ = ~obj.Success;
        end
        function draw(obj,p)      % draw the screen
            obj.Adapter.draw(p);  % call PolygonGraphic's draw()
        end
    end
end
----- End of TimeCounter.m -----
```

Each adapter has two functions, `analyze()` and `draw()`. These functions are called by `run_scene()` during each frame in turns. The first thing they do is to call the same functions in the underlying adapter (Line 18 & 23). You should not modify these lines, so as not to break the chain.

In `analyze()` of this adapter, we check whether the time that elapsed from the scene start passed `Duration` (Line 19). If it did, set Success true (or false otherwise). `continue_`, the return value of `analyze()`, determines whether we will keep running the scene in the next frame. Here `continue_` becomes false when Success is true, so the scene ends when the elapsed time is equal to or longer than `Duration`.

This adapter does not update any graphic, so we just call the underlying adapter's `draw()` and finish.

The input argument, p, in `analyze()` and `draw()` is an instantiation of the `RunSceneParam` class. It contains many useful variables and provides access to some other runtime functions within the adapter.

```
p.SceneStartTime:  trialtime when the scene started
p.SceneStartFrame: Frame number when the scene started
p.EventMarker:     Eventcodes assigned to this variable are stamped at the time when the next
                   frame is presented.

p.scene_time():    Time passed from the scene start
p.scene_frame():   Number of frames presented from the scene start

p.trialtime():     The same function that you call in the timing file.
p.goodmonkey():     The new 'nonblocking' option is especially useful when you call goodmonkey()
                   in an adapter
p.dashboard():     Display user texts on the control screen
```

MonkeyLogic Graphics Library (MGL)

To handle graphic objects directly as `PolygonGraphic` does (see ext\polygongraphic.m), you need to know how to use MGL (MonkeyLogic Graphics Library). The following is an example MGL code that shows a circle and a rectangle on the screen.

```
----- beginning of example code -----
mglcreatesubjectscreen(1,[0 0 0],[0 0 800 600],0);  % create the subject screen
mglcreatecontrolscreen([800 0 1200 300]);           % create the control screen

id = mgladdcircle([0 1 0; 1 0 0],[100 100]);        % add a circle
mglsetproperty(id,'origin',[400 300]);              % move the circle to the center
id2 = mgladdbox([1 1 1; 0 0 1],[150 150]);          % add a rectangle
mglsetproperty(id2,'origin',[400 300]);             % move the rectangle to the center
```

```
mglrendergraphic();                                % render the circle and the rectangle
mglpresent();                                      % present to the screen

mglactivategraphic([id id2],false);                % turn off the circle and the rectangle
mgldestroygraphic([id id2]);                       % destroy the objects

mgldestroycontrolscreen();                         % destroy the control screen
mgldestroysubjectscreen();                         % destroy the subject screen
----- end of example code -----
```

When you write your own adapter, the gray lines above are not necessary because NIMH ML takes care of them. What you need to do is 1) create objects (mgladdXXXX), 2) change their properties (mglsetproperty), 3) turn them on/off (mglactivategraphic) and 4) destroy them (mgldestroygraphic).

There are 9 functions that add graphic/sound objects. The sound object can be activated/deactivated by mglactivatesound and destroyed by mgldestroysound. To play it, use mglplaysound and mglstopsound.

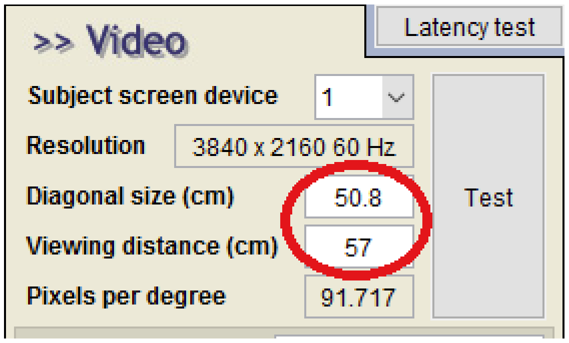
```
id = mgladdbitmap(filename);       % or mgladdbitmap(bitmap_info);
id = mgladdbox([edgecolor; facecolor],[width height]);
id = mgladdcircle([edgecolor; facecolor],[width height]);
id = mgladdline(color,numPoints);  % and mglsetproperty(id,'addpoint',[x1 y1; x2 y2; ...]);
id = mgladdmovie(filename);        % or mgladdmovie(frame_info);
id = mgladdpie([edgecolor; facecolor],[width height],start_angle,central_angle);
id = mgladdpolygon([edgecolor; facecolor],[width height],[x1 y1; x2 y2; ...]);
                                   % x & y: 0-1, normalized coordinates
id = mgladdtext(string);
id = mgladdsound(filename);        % or mgladdsound(y,fs);
```

All these functions return an object id, which you need for manipulating the property of the object. When the objects are created, they are active, by default (meaning they will be presented on the screen). If you don't want them to be shown, deactivate them by calling mglactivategraphic(id, false).

Each object has different properties. For the list of manipulatable properties, see mglsetproperty.m

7. Calibrating Eye/Joystick Signals

Calibration is a process of converting and aligning voltages of input signals to coordinates of visual object space. To do so, you first need to set the physical dimensions of the visual space correctly. Measure the diagonal size of the subject monitor and the distance between the monitor and the subject and put them in the ML main menu as below. Their units should be the same (let's all use centimeters, to avoid any confusion). ML then gives you a conversion factor, pixels per degree (91.717 in the figure shown below), which ML uses to calculate visual angles. You can use this number to convert the size of visual objects to visual angles. For example, a 320 x 240 movie is [320 240] / 91.717 = [3.4890 2.6167] degrees in visual angles.



Subject screen device: 1
Resolution: 3840 x 2160 60 Hz
Diagonal size (cm): 50.8
Viewing distance (cm): 57
Pixels per degree: 91.717

NIMH ML currently provides options to use **Raw Signal** and two calibration methods, **Origin & Gain** and **2-D Spatial Transformation**. The **Origin & Gain** requires calibration just for 2 fixation points and is much easier to use with untrained subjects. The **2-D Spatial Transformation** requires sampling voltages for at least 4 fixation points but can do more complex mapping by a projective transform. Both methods come with a tool that you can conveniently manipulate with the mouse. If you click yellow squares on the control screen, a fixation point will be shown in the corresponding location on the subject screen.

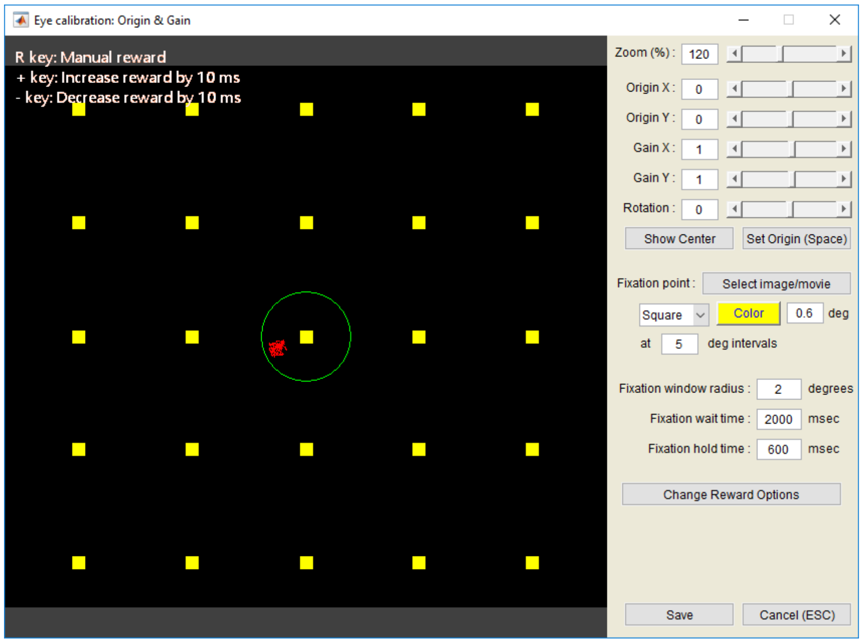
There are a few shortcut keys that you can use during calibration. These shortcuts work during the task as well.

- 'C' key: Brings the current eye position to the center of the screen.
- 'U' key: Cancels the previous 'C' key. You can undo multiple times.
- 'R' key: Delivers a manual reward.
- '-' key: Decreases the reward pulse length by 10 ms.
- '+' key: Increases the reward pulse length by 10 ms.

Raw Signal (Pre-calibrated)

Choose this option when the signals are calibrated already, or you want to get unaltered input. One volt in the calibrated signals should correspond to one degree’s displacement. If the signal is calibrated but the conversion ratio is not one to one, you can do further adjustment with the Origin & Gain method.

Origin & Gain



Eye calibration: Origin & Gain

R key: Manual reward
+ key: Increase reward by 10 ms
- key: Decrease reward by 10 ms

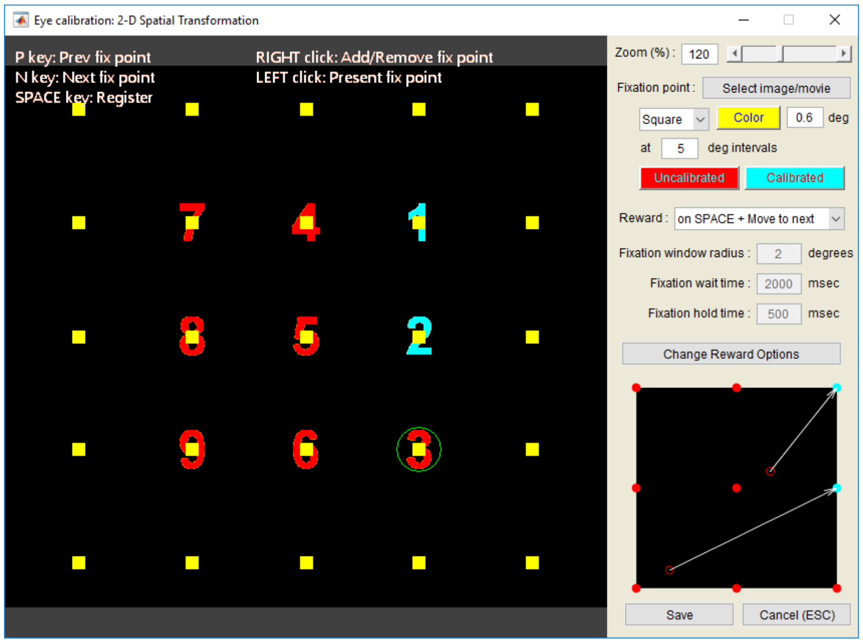
Zoom (%) : 120
Origin X : 0
Origin Y : 0
Gain X : 1
Gain Y : 1
Rotation : 0

Fixation point : Square 0.6 deg at 5 deg intervals

Fixation window radius : 2 degrees
Fixation wait time : 2000 msec
Fixation hold time : 600 msec

1. Click the "Show Center" button. It will show a fixation point at the center of the subject screen. Then wait until the subject look at it.
2. Click the "Set Origin" button while the subject is looking at the center. It will register the current voltage reading. Alternatively, you can click the fixation point (yellow square) at the center with the mouse left button and press the space bar (or ‘C’ key) to register the voltage.
3. Click one of the fixation points on the periphery. Increase (or decrease) X & Y gains if the subject's saccade undershoots (or overshoots).
4. Click the "Save" button. The "Cancel" button reverts any change that has been made.

2-D Spatial Transformation


Eye calibration: 2-D Spatial Transformation

P key: Prev fix point
N key: Next fix point
SPACE key: Register

RIGHT click: Add/Remove fix point
LEFT click: Present fix point

Zoom (%) : 120

Fixation point : Square, 0.6 deg at 5 deg intervals

Reward : on SPACE + Move to next

Fixation window radius : 2 degrees

Fixation wait time : 2000 msec

Fixation hold time : 500 msec

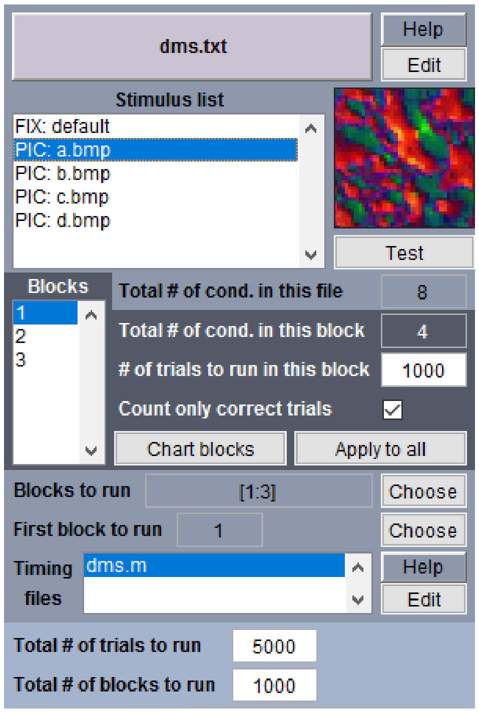
1. You can use only selected fixation points for calibration in this method. Selected points have numbers on them. To select/unselect fixation points, use the right mouse button. If you change the intervals of fixation points, then some numbers may not be on a fixation point any more. For those numbers, you can only unselect them.
2. Once you pick up all fixation points you want, you can turn them on/off one by one with the keyboard (N key & P key) or the mouse left-clicks.
3. Hit the space key while the subject is looking at the displayed fixation point. Then the voltage reading at that moment will be registered for the shown fixation point and its number will turn from red to blue.
4. You can choose how you want to reward the fixation behavior. The default option is to deliver reward at the time of the space key press and move to the next fixation point.
5. Repeat 2) and 3) for the entire selected fixation points a couple of times. Then click the "Save" button. Eye traces won't appear until there are at least 4 fixation points calibrated.
6. If the Reward option is switched to “On Fixation”, a reward is delivered automatically on fixation without a space key press. So, you can test the calibration without modifying the current setting.

8. Running a Task

Loading a Conditions/Userloop File

To start a task, either a conditions file or a userloop file should be loaded. A conditions file is a text file that describes what stimuli need to be presented with which timing script. A userloop file does the same but it is a MATLAB function, not a text file.

Both files can be loaded by clicking the [Load a conditions file] button. The file-open dialog shows *.m files only, by default, so you need to change the extension type to choose a userloop function. You can write the userloop filename in a text file and select that text file instead, to make this step simpler.



Upon loading a conditions/userloop file, some menu options are populated and activated, such as the “Blocks” pane and the “Blocks to run” button. These options are important to control the task flow.

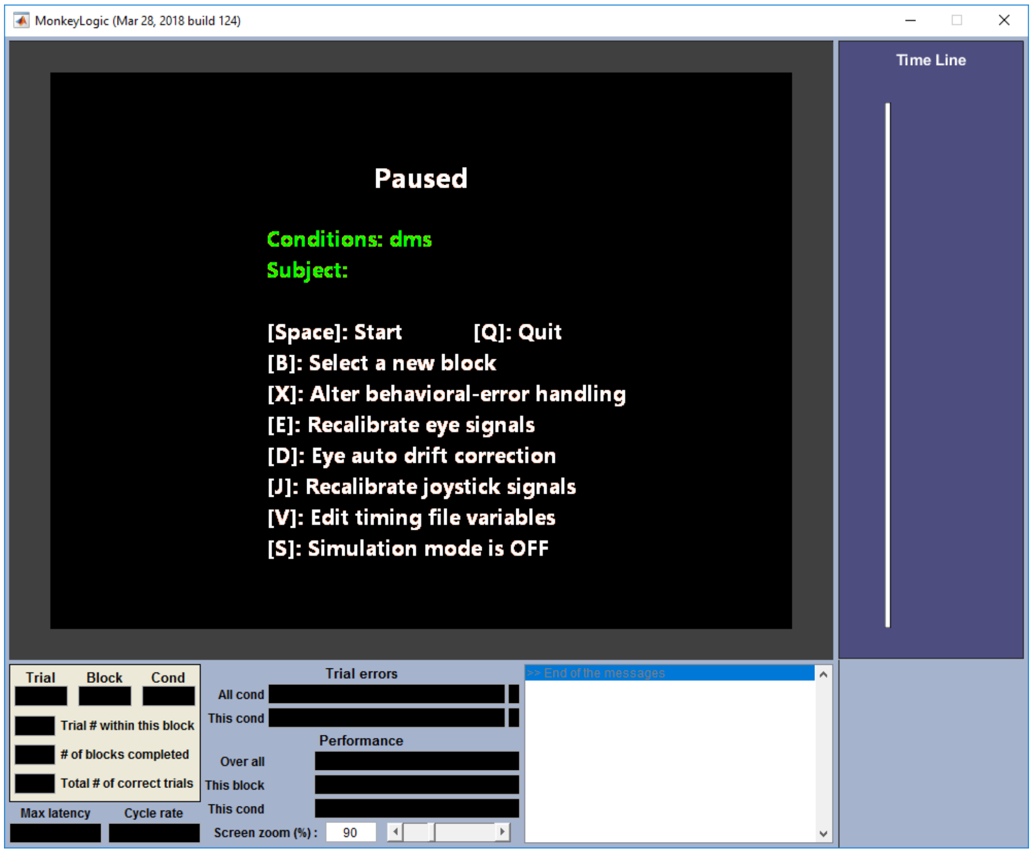
If you click on a different block number in the “Blocks” pane, “Total # of conditions in this block”, “# of trials to run in this block” and “Count only correct trials change” change accordingly and you can customize how many trials to run in each block.

“Blocks to run” and “First block to run” allow you to run only a subset of the defined blocks and choose which blocks to run very first.

“Total # of trials to run” and “Total # of blocks to run” let you decide when the task will stop. The task will be paused whichever between them comes first.

Additionally you should configure the size and resolution of the subject screen (see **Calibrating Eye/Joystick Signals**) and the DAQ device (see **DAQ board setting** and **Analog Input Ground Configuration**), to run the experiment properly.

Pause Menu



MonkeyLogic (Mar 28, 2018 build 124)

Paused

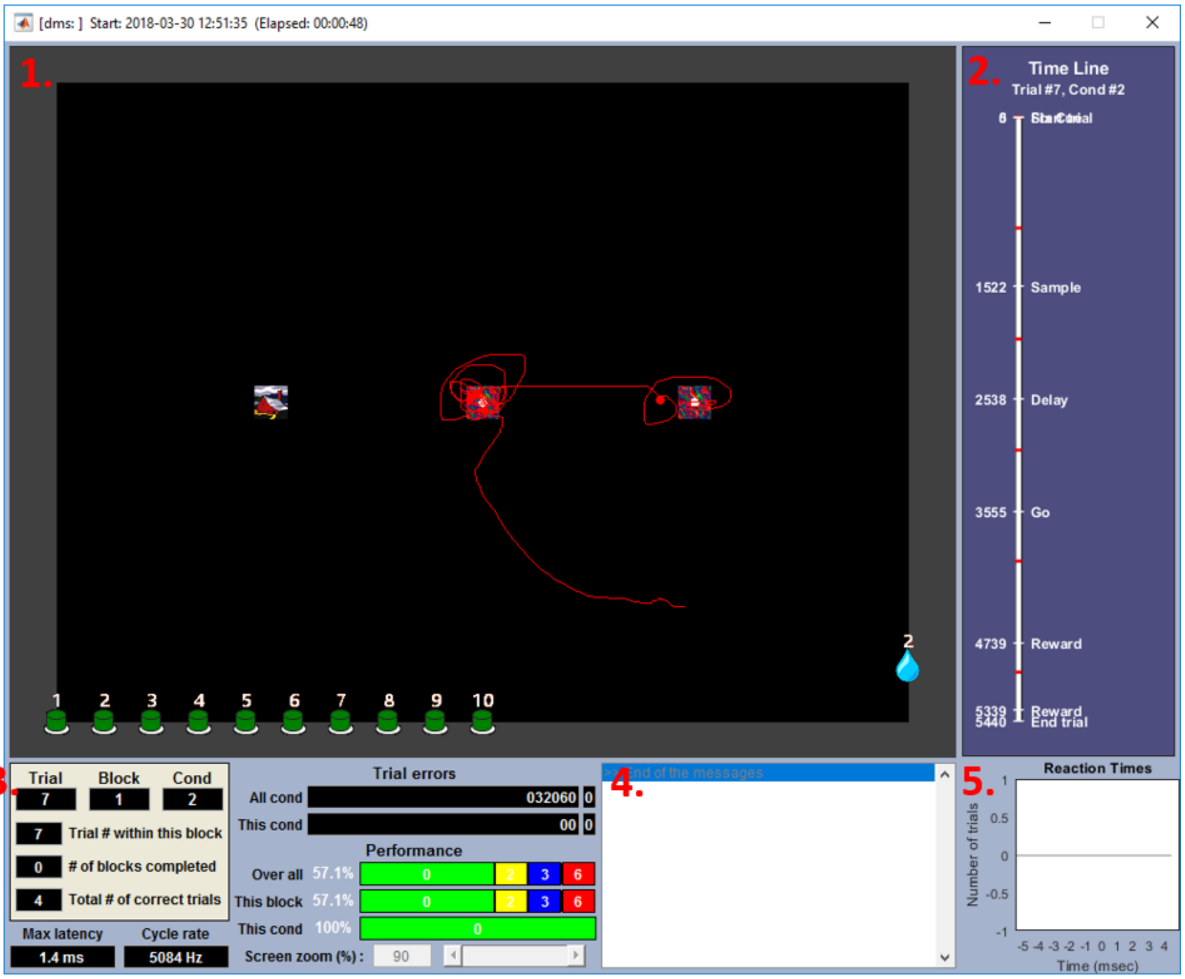
Conditions: dms
Subject:

[Space]: Start [Q]: Quit
[B]: Select a new block
[X]: Alter behavioral-error handling
[E]: Recalibrate eye signals
[D]: Eye auto drift correction
[J]: Recalibrate joystick signals
[V]: Edit timing file variables
[S]: Simulation mode is OFF

This pause menu appears when the RUN button is hit or when the task is paused. In this menu, the following options are available. You can use hotkeys or mouse-click to select an option.

- [Space] key: Start/resume the task
- [Q] key: Quit the task
- [B] key: Select a new block
- [X] key: Alter behavioral-error handling
- [E] key: Recalibrate eye signals
- [D] key: Eye auto drift correction
- [J] key: Recalibrate joystick signals
- [V] key: Edit timing file variables
- [S] key: Simulation mode On/Off

Control Screen


[dms:] Start: 2018-03-30 12:51:35 (Elapsed: 00:00:48)

Trial	Block	Cond
7	1	2

7	Trial # within this block
0	# of blocks completed
4	Total # of correct trials

Max latency	Cycle rate
1.4 ms	5084 Hz

Trial errors

All cond	032060	0
This cond	00	0

Performance

Over all	57.1%	0	2	3	6
This block	57.1%	0	2	3	6
This cond	100%	0			

Screen zoom (%) : 90

>> End of the messages

1. **Replica of the subject screen:** This panel displays extra information that is available for the experimenter only, such as the real-time input state, the location and size of fixation windows, reward delivery, etc., in addition to the stimuli presented to the subject. You can display user-defined dynamic texts during trials here (see the **dashboard** runtime function).
2. **Time line of events**
3. **Trial counts and results:** Trial results are number-coded as in the **trialerror** function. The zoom level is adjustable while the task is paused.

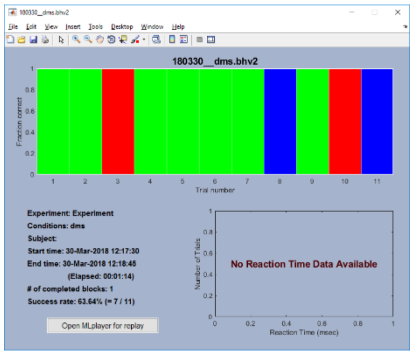
0: Correct	5: Early Response
1: No Response	6: Incorrect Response
2: Late Response	7: Lever Break
3: Break Fixation	8: Ignored
4: No Fixation	9: Aborted

4. **User text/warning panel:** Texts sent by the **user_text** and **user_warning** functions are displayed here. They are updated at the end of each trial, whereas the texts sent by **dashboard** are shown up immediately.
5. **User plot:** This figure can be used to show the result of online behavior analysis. Users can register their own functions in the main menu. If no function is registered, a default reaction time graph will be drawn here.

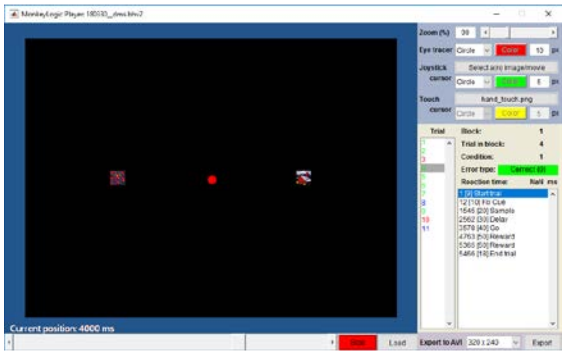
During trials, hotkeys like the following are available.

- 'ESC' key: Pauses the task
- 'C' key: Brings the current eye position to the center of the screen.
- 'U' key: Cancels the previous 'C' key. You can undo multiple times.
- 'R' key: Delivers a manual reward. The initial pulse duration is 100 ms.
- '-' key: Decreases the reward pulse length by 10 ms.
- '+' key: Increases the reward pulse length by 10 ms.

Behavioral Summary & mlplayer



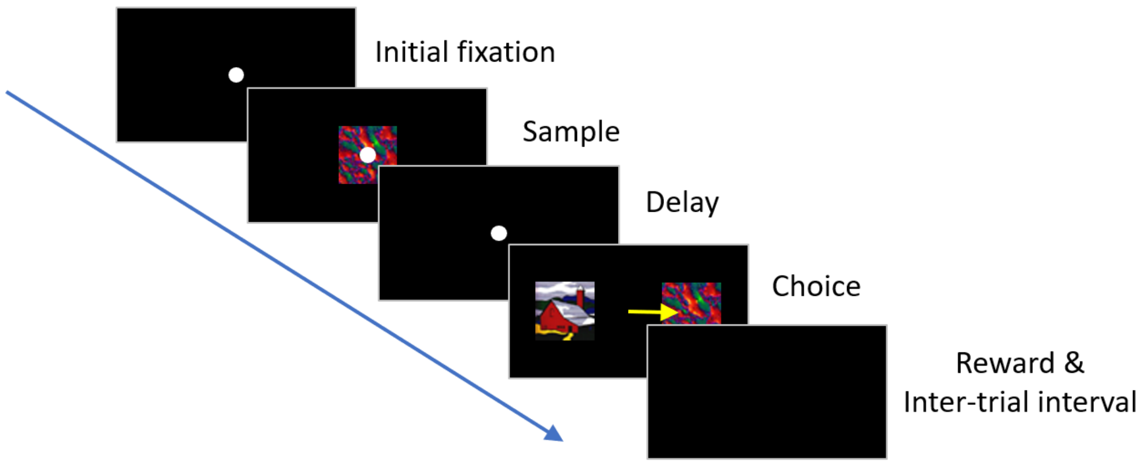
Experiment: Experiment
Conditions: dms
Subject:
Start time: 30-Mar-2018 12:17:30
End time: 30-Mar-2018 12:18:45
(Elapsed: 00:01:14)
# of completed blocks: 1
Success rate: 63.64% (= 7 / 11)



Zoom (%) 30
Eye tracer Circle 10 px
Joystick cursor Circle 5 px
Touch cursor Circle 5 px
Block: 1
Trial in block: 4
Condition: 1
Error type: Correct (0)
Reaction time: NaN ms
1 [9] Start trial
12 [10] Fix Cue
1545 [20] Sample
2562 [30] Delay
3578 [40] Go
4763 [50] Reward
5365 [50] Reward
5466 [18] End trial
Current position: 4000 ms
Export to AVI 320 x 240

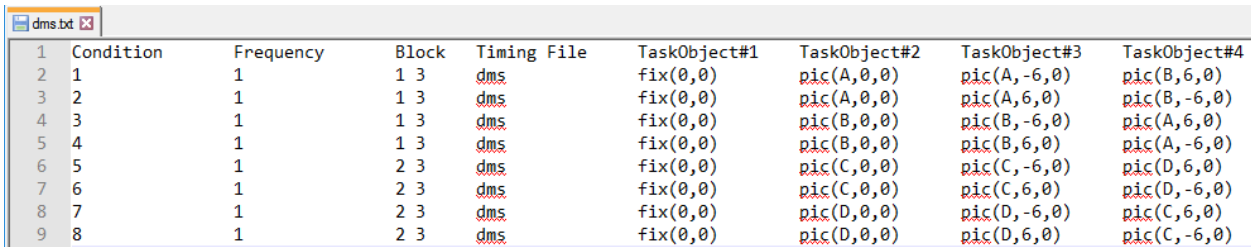
When the task is finished, a figure that summarizes behavioral performance is popped up (left figure). You can launch a trial replay tool, MonkeyLogic player (right figure), from this behavior summary and export each trial as a video clip. The summary figure and the player can be re-opened afterwards with the **behaviorsummary** and **mlplayer** functions.

9. Example: Delayed Match-to-Sample Task



A delayed match-to-sample (DMS) task requires a subject to remember the sample stimulus and identify it from a set of stimuli presented subsequently. In this version of the DMS task, a trial begins with an eye fixation. When the subject successfully fixates on the white circle shown at the center of the screen, an image (“sample”) is displayed briefly, turned off and followed by a delay period. At the end of the delay period, two images (“sample” and “distractor”) are presented on both sides of the screen and the subject is required to indicate her/his choice by making a saccade eye movement to the chosen target. If the choice is correct, a reward is delivered. Then, an inter-trial interval begins.

DMS Conditions File



Condition	Frequency	Block	Timing File	TaskObject#1	TaskObject#2	TaskObject#3	TaskObject#4
1	1	1 3	dms	fix(0,0)	pic(A,0,0)	pic(A,-6,0)	pic(B,6,0)
2	1	1 3	dms	fix(0,0)	pic(A,0,0)	pic(A,6,0)	pic(B,-6,0)
3	1	1 3	dms	fix(0,0)	pic(B,0,0)	pic(B,-6,0)	pic(A,6,0)
4	1	1 3	dms	fix(0,0)	pic(B,0,0)	pic(B,6,0)	pic(A,-6,0)
5	1	2 3	dms	fix(0,0)	pic(C,0,0)	pic(C,-6,0)	pic(D,6,0)
6	1	2 3	dms	fix(0,0)	pic(C,0,0)	pic(C,6,0)	pic(D,-6,0)
7	1	2 3	dms	fix(0,0)	pic(D,0,0)	pic(D,-6,0)	pic(C,6,0)
8	1	2 3	dms	fix(0,0)	pic(D,0,0)	pic(D,6,0)	pic(C,-6,0)

The above figure is an example conditions file necessary to implement the task. Conditions 1-4 are included in Block 1 and Conditions 5-6 are included in Block 2. In Block 4, all 8 conditions can be run. Each condition defines 4 stimuli (fixation cue, sample, match and distractor) and they are controlled by the “dms" timing script.

DMS Timing Script

The timing script written for this example is under the "task\runtime v1\1 dms" task. The script shows how you can present stimuli and monitor behavioral response. In the runtime library version 1, this is done by the **toggleobject** and **eyejoytrack** functions.

```
% initial fixation:
toggleobject(fixation_point, 'eventmarker',10);
ontarget = eyejoytrack('acquirefix', fixation_point, fix_radius, wait_for_fix);
if ~ontarget
    toggleobject(fixation_point);
    trialerror(4);  % no fixation
    return
end
ontarget = eyejoytrack('holdfix', fixation_point, hold_radius, initial_fix);
if ~ontarget
    toggleobject(fixation_point);
    trialerror(3);  % broke fixation
    return
end
```

In the above code, toggleobject turns on "fixation point" and marks a code of 10 in the data file with a timestamp. Then eyejoytrack waits until the eye fixation is made. If the fixation is not made within the "wait_for_fix" time (i.e., ontarget is 0), we turn off the fixation point, record the trial result ("no fixation") and return. Otherwise, the task proceeds to the next step and another eyejoytrack checks if the fixation is held.

The same job can be scripted in a different way with the runtime library version 2. In the runtime v2, the toggleobject and eyejoytrack functions are replaced with create_scene and run_scene, like the following. The entire code of the runtime v2 example is in the "task\runtime v2\1 dms with new runtime" directory.

```
% scene 1: fixation
fix1 = SingleTarget(eye_);
fix1.Target = fixation_point;
fix1.Threshold = fix_radius;
wth1 = WaitThenHold(fix1);
wth1.WaitTime = wait_for_fix;
wth1.HoldTime = initial_fix;
scene1 = create_scene(wth1, fixation_point);
run_scene(scene1, 10);
if ~wth1.Success
    if wth1.Waiting
        trialerror(4);
    else
        trialerror(3);
    end
    return
end
```

10. NI Multifunction I/O Device

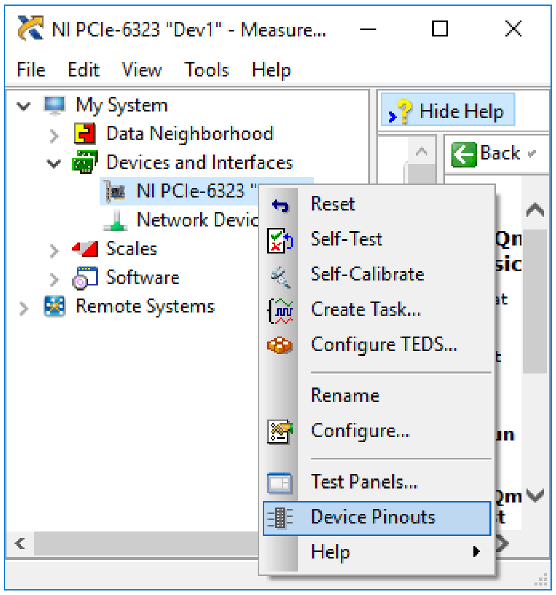
Selecting a DAQ Board

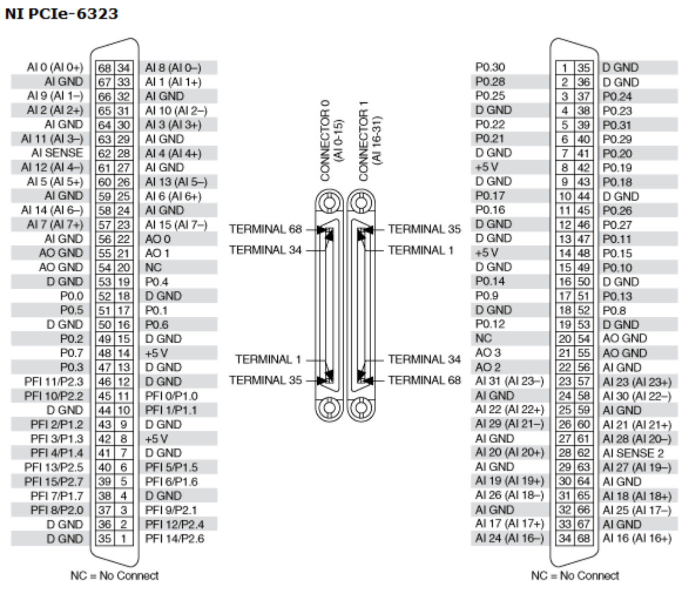
NIMH ML works with any NI board that is supported by the NI-DAQmx driver. There is no need to buy a new board, if you already have one from your old systems. If you buy a new one, pick up the one that can handle all your I/O needs in one board. It will save the overall cost greatly. Usually new products using the PCIe bus are cheaper than old PCI-type boards. The following table lists some NI boards that are suitable for general I/O requirements.

Model	Analog Input (SE/DIFF)	Analog Output	Digital I/O
PCI-6221	16 / 8	2	24 (P0: 8, P1: 8, P2: 8)
PCI-6229	32 / 16	4	48 (P0: 32, P1: 8, P2: 8)
PCIe-6320	16 / 8	0	24 (P0: 8, P1: 8, P2: 8)
PCIe-6321	16 / 8	2	24 (P0: 8, P1: 8, P2: 8)
PCIe-6323	32 / 16	4	48 (P0: 32, P1: 8, P2: 8)

Device Pinouts

To connect external devices to the NI board, you need to know which pin is mapped to which signal. You can find this information in the product's datasheet (PDFs on the web) or by right-clicking on installed devices in the NI Measurement & Automation Explorer (NI MAX) software (Windows help file).





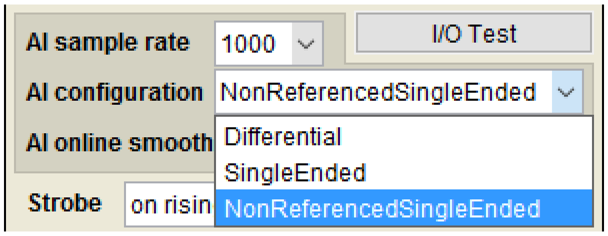
Inputs from external devices can be connected to NI boards via NI terminal blocks. Unshielded screw terminal blocks, such as CB-68LP and CB-68LPR, are low-cost and good for designing a custom interface box of your own. Or you can choose BNC terminal blocks, like BNC-2090A, if you want some ready-made solution. For the details, please refer to the NI DAQ Multifunction I/O Accessory Guide.

Analog Input Ground Configuration

To measure analog voltage signals accurately, you need to know whether the signals are ground-referenced or floating and configure the data acquisition device (i.e., NI board) accordingly. NI boards require different wiring schemes for different ground configuration, so you should be aware how the signal sources are connected to your board. There are typically three ground configuration modes available for NI devices; differential (DIFF), referenced single-ended (RSE) and nonreferenced single-ended (NRSE).

If your signal sources are referred to an absolute voltage reference, such as earth or building ground, you can say your signals are ground-referenced and use the RSE mode. However, in a typical lab environment where many devices run on custom power supplies or batteries, it is likely that your signals are floating sources which are not tied to a fixed reference. Then, you should use DIFF or NRSE. For more information, please refer to this online document, Field Wiring and Noise Considerations for Analog Signals.

Below are the instructions how you should connect the leads from your signal sources to NI devices, depending on the ground configuration of your choice. You also need to change the [AI configuration] option on the ML main menu.



Differential mode (DIFF)

In the differential mode, measuring a signal requires connecting one lead of the signal source to an AI channel pin (AI 0, AI 1, ...) and the other lead to a channel larger by 8. For example, if we measure a signal on channel 0, one lead goes to AI 0 (Ch 0+) and the other to AI 8 (Ch 0-). This mode can deliver more accurate measurements with less noise but takes twice as many channels of the NI board as the other modes.

Referenced Single-Ended mode (RSE)

In this mode, the measurement is made with respect to the common-mode voltage, for example, the earth ground of the wall outlets. For this configuration, one lead of each signal source should be connected to an AI channel pin (AI 0, AI 1, ...) and the other lead to AI GND.

Nonreferenced Single-Ended mode (NRSE)

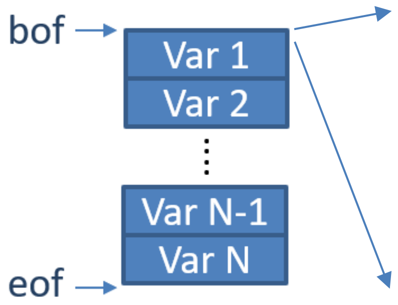
If you have many custom devices to connect and you are not sure whether they are properly grounded to earth, Nonreferenced Single-Ended should be the mode of your choice. In this mode, one lead must be connected to an AI channel pin (AI 0, AI 1, ...) and the other lead to AI SENSE.

11. Appendix

BHV2 Binary Structure

BHV2 is a custom binary format designed to store MATLAB variables. It has a very simple structure that can be read with a recursive algorithm.

BHV2 has no file header and just contains the contents of variables. Each variable block starts with 6 fields like the following. The 1st, 3rd, 5th fields Indicates the lengths of the 2nd, 4th, 6th fields, respectively.



Field	Type	Length
Length of Variable Name (LN)	uint64	1
Variable Name	char*1	LN
Length of Variable Type (LT)	uint64	1
Variable Type	char*1	LT
Dimension of Variable (DV)	uint64	1
Size of Variable	uint64	DV

If the variable type is one of the MATLAB primitive data types (char, integers, single, double, logical), then the content of the variable follows those 6 fields in **column-major order**. For example, if A = rand(2,2), the byte order of A will be like this.

1	A	6	double	2	[2 2]	A(1,1)	A(2,1)	A(1,2)	A(2,2)

If the variable type is **struct**, there is one more field of uint64 that indicates the number of fields. Then the first field of the first struct array starts.

```
  ex) A = [struct('a',1,'b','xyz') struct('a',9,'b','')];  % the number
of fields is 2; a & b.
```

1	A	6	struct	2	[1 2]	2	A(1).a	A(1).b	A(2).a
A(2).b									

If the variable type is **cell**, the cells of the cell array comes in column-major order.

```
ex) A = cell(3,2);
```

1	A	4	cell	2	[3 2]	A{1,1}	A{2,1}	A{3,1}	A{1,2}
A{2,2}	A{3,2}								

Byte order of a struct

```
A(1).a = [1 2 3];
A(1).b = 'xyz';
A(2).a = [5 6; 7 8];
A(2).b = '';
```

```
1         [1x1 uint64]  % length('A')
A         [1x1 char]    % struct name
6         [1x1 uint64]  % length('struct')
struct    [1x6 char]    % variable type
2         [1x1 uint64]  % dimension of variable
[1 2]     [1x2 double]  % size of struct
2         [1x1 uint64]  % number of fields in A
1         [1x1 uint64]  % length('a')
a         [1x1 char]    % name of the first field
6         [1x1 uint64]  % length('double')
double    [1x6 char]    % variable type
2         [1x1 uint64]  % dimension of variable
[1 3]     [1x2 double]  % size of variable
1 2 3     [1x3 double]  % content of the first field
1         [1x1 uint64]  % length('b')
b         [1x1 char]    % name of the second field
4         [1x1 uint64]  % length('char')
char      [1x4 char]    % variable type
2         [1x1 uint64]  % dimension of variable
[1 3]     [1x3 double]  % size of variable
xyz       [1x3 char]    % content of the second field
1         [1x1 uint64]  % length('a')
a         [1x1 char]    % name of the first field
6         [1x1 uint64]  % length('double')
double    [1x6 char]    % variable type
2         [1x1 uint64]  % dimension of variable
[2 2]     [1x2 double]  % size of variable
5 7 6 8   [2x2 double]  % content of the first field
1         [1x1 uint64]  % length('b')
b         [1x1 char]    % name of the second field
4         [1x1 uint64]  % length('char')
char      [1x4 char]    % variable type
2         [1x1 uint64]  % dimension of variable
[0 0]     [1x2 double]  % size of variable
~~''        [0x0 char]    % content of the second field~~
(end)
```

The last byte in the above example does not exist since its content is blank. Note that the content of A(2).a is written as [5 7 6 8], not [5 6 7 8], since arrays are in column major order in MATLAB.

Byte order of a cell

```
A = cell(2,2);
A{1,1} = [1 2 3];
A{1,2} = 'xyz';
A{2,1} = [5 6; 7 8];
A{2,2} = '';
```

```
  1          [1x1 uint64]   % length('A')
  A          [1x1 char]     % cell array name
  4          [1x1 uint64]   % length('cell')
  cell       [1x4 char]     % variable type
  2          [1x1 uint64]   % dimension of variable
  [2 2]      [1x2 double]   % size of cell array
  0          [1x1 uint64]   % A{1,1} doesn't have name
  ~~''         [0x0 char]     % no name~~
  6          [1x1 uint64]   % length('double')
  double     [1x6 char]     % variable type
  2          [1x1 uint64]   % dimension of variable
  [1 3]      [1x2 double]   % size of variable
  1 2 3      [1x3 double]   % content of the A{1,1}
  0          [1x1 uint64]   % A{2,1} doesn't have name
  ~~''         [0x0 char]     % no name~~
  6          [1x1 uint64]   % length('double')
  double     [1x6 char]     % variable type
  2          [1x1 uint64]   % dimension of variable
  [2 2]      [1x2 double]   % size of variable
  5 7 6 8    [2x2 double]   % content of A{2,1}
  0          [1x1 uint64]   % A{1,2} has no name
  ~~''         [0x0 char]     % no name~~
  4          [1x1 uint64]   % length('char')
  char       [1x4 char]     % variable type
  2          [1x1 uint64]   % dimension of variable
  [1 3]      [1x2 double]   % size of variable
  xyz        [1x3 double]   % content of A{1,2}
  0          [1x1 uint64]   % A{2,2} has no name
  ~~''         [0x0 char]     % no name~~
  4          [1x1 uint64]   % length('char')
  char       [1x4 char]     % variable type
  2          [1x1 uint64]   % dimension of variable
  [0 0]      [1x2 double]   % size of variable
  ~~''         [0x0 char]     % content of A{2,2}~~
(end)
```

Again, the fields that have any 0-sized dimension are not written to the file. And not only a double matrix (A{2,1}) but also a cell array ('A' itself) is arranged in column major order.

HDF5 Implementation

Creating and reading .h5 files are implemented with MATLAB's low-level HDF5 access functions. In BHV2, the name, type and size of each variable are stored in the 6 fields that come at the beginning of each variable block. In H5, those fields are stored as attributes of datasets or groups.

The HDF5 implementation in MATLAB R2014b or earlier does not allow to create a 0-sized dataspace. So the H5 files created in those versions contains a single 0, even when the variable is empty. If the size attribute of a variable is 0, you should discard its value, 0, when you read the variable in your application.

The MATLAB primitive data types are stored in datasets. The struct and cell arrays are created as groups of the fields or cells.

H5 can be read with any common HDF5 tool, like HDFView (https://support.hdfgroup.org/products/java/hdfview/).

Refer to mlhdf5.m for implementation details.

TrialRecord Structure

TrialRecord is a MATLAB struct that contains information about the trial history. This structure has the following fields that are updated every trial.

Fields of TrialRecord	Remarks
CurrentTrialNumber	The currently executing trial number, consecutively numbered from the start of the session *(scalar)*
CurrentTrialWithinBlock	The currently executing trial number, consecutively numbered from the start of the current block *(scalar)*
CurrentCondition	The condition number of the current trial *(scalar)*
CurrentBlock	The block number of the current trial *(scalar)*
CurrentBlockCount	The total number of blocks played thus far, including the current one *(scalar)*
CurrentConditionInfo	A struct created from the “info” column of the conditions file
CurrentConditionStimulusInfo	Information about the stimuli used in the current trial *(cell array, one cell per TaskObject)*
ConditionsPlayed	The list of conditions played since the start of the session *(a vector of length (CurrentTrialNumber - 1))*
ConditionsThisBlock	The list of conditions which are available for selection from within the current block *(vector of variable length)*
BlocksPlayed	The list of blocks played since the start of the session *(a vector of length (CurrentTrialNumber - 1))*
BlockCount	The number of blocks played, as of any given trial thus far *(a vector of length (CurrentTrialNumber - 1))*
BlockOrder	The order of blocks played thus far, including the current block *(vector)*
BlocksSelected	The list of blocks available, as selected from the main menu *(vector)*
TrialErrors	The list of behavioral errors for every trial played so far *(a vector of length (CurrentTrialNumber - 1))*
ReactionTimes	The list of reaction times for every trial played so far *(a vector of length (CurrentTrialNumber - 1))*
LastTrialAnalogData	A structure containing the fields **EyeSignal** and **Joystick**, with the x- and y-vectors of the last trial's analog signals
LastTrialCodes	A structure containing the fields CodeNumbers and CodeTimes, containing vectors corresponding to each
Editable	A struct of “editable” variables defined in the timing script

While the above fields are automatically updated by NIMH ML and not editable, TrialRecord has a few other user-editable fields for the task flow control and the data exchange.

Field of TrialRecord	Default Value	Remarks
Pause	false	If true, the task will be pause after the current trial.
Quit	false	If true, the task will be quitted after the current trial.
DrawTimeLine	true	If false, the Time Line panel of the control screen is NOT updated after each trial. This option is useful to suppress the output, when there are too many event codes to draw.
MarkSkippedFrames	false	If true, eventcode 13 will be marked in case of frame skipping.
DiscardSkippedFrames	true	If true, missing frames of movies are discarded when skipped frames occur.
User	[]	This is a field that users can store temporary variables that they want to pass to other functions across trials.

TrialRecord is used as input to many functions during tasks, such as timing scripts, condition-selection functions, block-selection functions, block-change functions, user-plot functions and user-generated image functions. If you need to deliver additional input to those functions, you can do so by creating new fields under TrialRecord.User and assign values there.

Structure of Conditions File

Conditions files are text files which specify each possible combination of stimuli ("TaskObjects") within a trial. Each of these stimulus combinations comprising a unique trial type is referred to as a "condition." During experiment execution, conditions are selected to run as determined by the options set in the main menu. They can be grouped into blocks and selected collectively. The rule of selecting/switching blocks can also be determined on the menu (see **Blocks** and **Block-change function**).

Below is an example of a conditions file for a delayed match-to-sample task using a total of 4 picture stimuli (A, B, C and D). A conditions file consists of a header followed by consecutively numbered conditions. All columns are tab-delimited (multiple tabs without text will be reduced to one, and no intervening blank columns are permitted).

Condition	Info	Frequency	Block	Timing File	TaskObject#1	TaskObject#2	TaskObject#3	TaskObject#4
1	'samp','A','match',-1	1	1 3	dms	fix(0,0)	pic(A,0,0)	pic(A,-4,0)	pic(B,4,0)
2	'samp','A','match',1	1	1 3	dms	fix(0,0)	pic(A,0,0)	pic(A,4,0)	pic(B,-4,0)
3	'samp','B','match',-1	1	1 3	dms	fix(0,0)	pic(B,0,0)	pic(B,-4,0)	pic(A,4,0)
4	'samp','B','match',1	1	1 3	dms	fix(0,0)	pic(B,0,0)	pic(B,4,0)	pic(A,-4,0)
5	'samp','C','match',-1	1	2 3	dms	fix(0,0)	pic(C,0,0)	pic(C,-4,0)	pic(D,4,0)
6	'samp','C','match',1	1	2 3	dms	fix(0,0)	pic(C,0,0)	pic(C,4,0)	pic(D,-4,0)
7	'samp','D','match',-1	1	2 3	dms	fix(0,0)	pic(D,0,0)	pic(D,-4,0)	pic(C,4,0)
8	'samp','D','match',1	1	2 3	dms	fix(0,0)	pic(D,0,0)	pic(D,4,0)	pic(C,-4,0)

The *Info* column is here being used to pass labels to the timing file about which image is being used for the sample and where the matching image is being displayed; this column is intended to make deciphering the conditions easier for the user, and does not affect actual task execution. Users can access this information in the timing script, like "`Info.samp`" and "`Info.match`". This column is optional and you can delete it from the header, if you don't need it.

The *Frequency* column gives the weight or likelihood of that particular condition being selected randomly relative to other conditions. For example, if a condition has a relative frequency of 3, it is as if that condition is listed 3 times; that is, it has 3 times the chance of being selected as a trial with a relative frequency of 1, if conditions are being selected at random within each block.

In the *Block* column, the numbers correspond to those blocks in which each condition can appear. Here, for instance, block 1 uses only images A and B, block 2 uses C and D, and all images are used in block 3. Therefore, if the user chooses to run only block 2 from the main menu (or if that block is selected on-line during task execution according to pre-specified criteria), only conditions 5-8 will constitute the pool of possible conditions to run; running block 3, on the other hand, will allow all conditions to be placed into the selection pool.

The *Timing File* refers to the Matlab m-script which calls up each stimulus at the appropriate instant and checks for fixation, target acquisition, etc. Each condition can be associated with a different timing file, if desired.

TaskObjects are identified by their columnar locations (i.e., TaskObject numbers). They consist of three-letter symbols (FIX**,** PIC, MOV, CRC, SQR, SND, STM, TTL and GEN) and parameters for their generation or display (see **TaskObjects** for details).

TaskObjects

Type	Syntax	Remarks
Fixation point[1]	fix(Xdeg, Ydeg)	• **Xdeg** and **Ydeg:** XY positions in degrees[2]

Static image[1]	pic(filename, Xdeg, Ydeg)
	pic(filename, Xdeg, Ydeg, colorkey)
	pic(filename, Xdeg, Ydeg, Wpx, Hpx)
	pic(filename, Xdeg, Ydeg, Wpx, Hpx, colorkey)
		• **filename:** BMP, GIF, JPG, TIF or PNG
		• **Xdeg** and **Ydeg:** in degrees[2]
		• **colorkey:** a color [R G B] which will be treated as transparent
		• **Wpx** and **Hpx:** optional resizing parameters (width and height in pixels)

Movie[1]	mov(filename, Xdeg, Ydeg)	• **filename:** AVI or MPG • **Xdeg** and **Ydeg:** in degrees[2]
Circle[1]	crc(radius, RGB, fill, Xdeg, Ydeg)	• **radius:** in degrees • **RGB:** a triplet [R G B] with values 0-1 • **fill:** 0 (outline) or 1 (filled) • **Xdeg** and **Ydeg:** in degrees[2]
Square[1]	sqr(size, RGB, fill, Xdeg, Ydeg)	• **size:** 1 element (square) or 2 (rectangle) in degrees • **RGB:** a triplet [R G B] with values 0-1 • **fill:** 0 (outline) or 1 (filled) • **Xdeg** and **Ydeg:** in degrees[2]

Sound	snd(filename)
	snd(*sin*, duration, frequency)
		• **filename:** WAV or MAT[3]
		• ***sin*** is to be typed literally.
		• **duration:** in seconds
		• **frequency:** in Hertz

Stimulation	stm(port, datasource)
	stm(port, datasource, retriggerable)
		• **port:** Stimulation # on the main menu I/O panel
		• **datasource:** MAT[3]
		• **retriggerable:** 0 (can be triggered only once) or 1 (multiple times)[4]

TTL pulse	ttl(port)	• **port:** TTL # on the main menu I/O panel

User-generated Pic or Mov[1]	gen(function_name)
	gen(function_name, Xdeg, Ydeg)
		• **function_name:** a user-provided MATLAB function[5]
		• **Xdeg** and **Ydeg:** in degrees[2]

1. For visual stimuli, an object with a smaller number will layer atop those with larger numbers. For example, when TaskObject#1 and TaskObject#2 are presented at the same location, TaskObject#1 will appear in front of TaskObject#2.

2. Relative to the screen center. + is up & right.

3. The MAT files must contain two variables, “y” and “fs”, for waveform and frequency, respectively.

4. When the “retriggerable” flag of an STM object is 1, stopping the stimulation will take a slightly longer time, to reload the waveform.

5. The GEN user function can take one of the following prototypes.

```
imdata = gen_func(TrialRecord);
imdata = gen_func(TrialRecord, MLConfig);
[imdata, info] = gen_func(___);
[imdata, Xdeg, Ydeg] = gen_func(___);
[imdata, Xdeg, Ydeg, info] = gen_func(___);
```

The GEN function takes the **TrialRecord** structure as input and can optionally take the **MLConfig** structure as the second argument.

imdata can be a filename or a matrix of one of the following dimensions.

X-by-Y: gray-scale image
X-by-Y-by-3: RGB image
X-by-Y-by-4: ARGB image (A: alpha channel)
X-by-Y-by-3-by-N: RGB movie (N: # of frames)
X-by-Y-by-4-by-N: ARGB movie (A: alpha channel, N: # of frames)

If **Xdeg** and **Ydeg** are not given in the conditions file, they can be provided from the GEN function. By default, they are both 0.

By adding new fields to the **info** structure, users can deliver extra information about the GEN stimulus to the timing script or other user functions. There are a few reserved field names as below.

info.Colorkey: a value of a color, [R G B], which will be treated as transparent
info.TimePerFrame: intervals of movie frames; in milliseconds
info.Looping: make movies repeated when the last frame is reached

The **info** structure can be accessed in the timing script or other user functions like the following.

```
TrialRecord.CurrentConditionsStimulusInfo(TaskObject#).MoreInfo
            or
StimulusInfo(TaskObject#).MoreInfo        % for timing script only
```

Runtime Library Functions for Timing Script

You can use the following runtime library functions as well as any valid MATLAB expression, when you write timing scripts. For the updated details of each function, see the timing script manual (doc\runtimefunctions.html in the ML directory).

Runtime functions

- **bhv_code**
- **bhv_variable**
- **dashboard**
- **editable**
- **escape_screen**
- **eventmarker**
- **eye_position**
- **get_analog_data**
- **get_movie_duration**
- **get_sound_duration**
- **getkeypress**
- **goodmonkey**
- **hotkey**
- **idle**
- **joystick_position**
- **mouse_position**
- **reposition_object**
- **rewind_movie**
- **rewind_sound**
- **set_bgcolor**
- **set_iti**
- **showcursor**
- **trialerror**
- **trialtime**
- **user_text**
- **user_warning**

Version 1 specific

- **eyejoytrack**
- **set_frame_event**
- **set_frame_order**
- **set_object_path**
- **toggleobject**

Version 2 specific

- **create_scene**
- **run_scene**