PSYCHTOOLBOX

A BRIEF GUIDE TO START PROGRAMMING EXPERIMENTS IN PSYCHOLOGY

María Hernández Lorca
Almudena Capilla

UAM EDICIONES

EDICIONES DE LA UNIVERSIDAD AUTÓNOMA DE MADRID
28049 Madrid
Teléfono 91 497 42 33
Fax 91 497 51 69
servicio.publicaciones@uam.es
www.uam.es/publicaciones

DOI: https://doi.org/10.15366/manual.psych2018
Diseño: Ana Palomo

PSYCHTOOLBOX

A BRIEF GUIDE TO START PROGRAMMING EXPERIMENTS IN PSYCHOLOGY

María Hernández Lorca
Almudena Capilla

INDEX

This work has been supported by a UAM 2016-17 Teaching Innovation Project (PS_1.16)

1. INTRODUCTION

Research in Psychology faces the challenge of understanding human behavior by means of experiments while the human participant is awake, conscious and, usually, responding to stimulation. To answer our research questions, we count on many different techniques to register behavioral data, but to be successful in this endeavor, we need to ensure that both stimulation and data collection of participant responses are carried out with high precision.

There is a broad range of software useful for these purposes available on the market. Among these, we favor and recommend the toolbox Psychtoolbox (PTB) to carry out the typical experimental tasks employed in Psychology, as well as in related fields such as Cognitive and Affective Neuroscience. The reasons for recommending PTB are many, but the most important is the fact that it covers the experimental needs in Psychology research: it is versatile in the presentation of stimuli, efficient in the programming, very accurate with stimulation (placement on the screen and presentation time), and very accurate with data collection (e.g. reaction times). Furthermore, PTB can be downloaded from their website (www.psychtoolbox.org) for free.

With this tutorial we aim to introduce the toolbox from the beginner's point of view, so that the reader gets familiar with the most common PTB programming commands and arguments, learns how to program a typical Psychology experiment, becomes capable of solving the most common problems one can encounter when beginning to program an experiment with PTB, and gets to know how and where to find the necessary information to overcome more complex problems.

PTB runs in MATLAB and GNU Octave, and most commands you need when programming an experiment are based on this language. This entire tutorial, the theoretical as well as the practical approach, is based on MATLAB, and we therefore recommend to have at least a basic knowledge of MATLAB language and functions before starting. In case you need some training, the 'Matlab for Psychologists' tutorial by Antonia Hamilton can be of great help (http://www.antoniahamilton.com/matlab.html).

This tutorial is divided into two sections: an introductory part to some of the PTB functions and their use, and a practical part where those functions are put into practice in a classic Psychology experiment. The first part covers three areas that are especially useful to psychological experimentation: 1) Related to the stimulation, and therefore, and for the most part, to what is shown on the screen. 2) Related to time, reaction times or durations: PTB has a very high degree of timing accuracy, which gives precise control over these important variables in psychological research. 3) Related to the participant's response, and therefore related to the information provided via the keyboard or any other response device (e.g., key presses).

The practical part illustrates how to program a typical experiment, step by step, with the code necessary to achieve the specific goal of each step. The result is a complete script for a short experiment that will cover the usual needs in Psychology such as presentation of pictures, response collection or reaction time recording, among many others.

The PTB webpage also contains demos and *how tos* that are very useful when learning specific functions or in case of needed clarifications or further information (see http://peterscarfe.com/ptbtutorials.html). A Google search with specific questions or problems can be very useful, and it is very likely that the problem one is experiencing has already been asked and answered in some online community.

Programming can be very discouraging and frustrating, but we entice our readers to be patient and persistent, use up all the resources that the online community so generously offers for solving doubts and never to lose faith and confidence in the ability to program well and solve any problems that might occur along the way.

2. PART 1: PTB FUNCTIONS AND THEIR USES

In a typical Psychology experiment, stimulation usually comes in the form of visual stimuli presented on a computer screen or via auditory stimuli. This chapter is about all the PTB functions that serve sensory stimulation, their use and specifications. Please note that these functions are only related to the presentation of the stimuli and decoding the participant's response and reaction time. To actually record and save the data we will need MATLAB code. The practical section will illustrate a script as a whole, and it will therefore include the code needed for collecting and saving the data.

We recommend that you try the examples that illustrate each function and play with them by changing parameters and experiment with the outcome. By doing this, you will gain a more real knowledge and comprehension of the function and will be more prepared to use it the way your experiment demands.

2.1 STIMULATION

2.1.1 Visual Stimulation

The `Screen` Function

The PTB function `Screen` controls what happens on the computer screen. The structure of this function is `Screen('command', arg1, arg2, ...)`. The command defines the instruction, and the arguments the specification to that instruction. To find out all the functions that `Screen` has, you can type `Screen` in the Command Window for a list. Similarly, to find the details on how to use the different instructions within the `Screen` function, we can use the help that is built in the toolbox. If the usual way to ask for help in MATLAB and any other PTB function is

```
>> help function
```

with `Screen` you call the help by typing in the Command Window

```
>> Screen command?
```

The response appears in the Command Window with the usage and specifications. Thus, if we type in the Command Window:

```
>> Screen FillRect?
```

the feedback in the Command Window is

```
Usage:

Screen('FillRect', windowPtr [,color] [,rect] )
```

This means that if we were to call the function `Screen` with the command `'FillRect'`, we would have to code it as it is shown. The arguments with no brackets [ ] around them must be filled in, whereas those that have brackets around can be left blank and PTB will use default values. If we were to specify values we would need to separate the arguments by a comma. Furthermore, if we wanted to specify an argument that is placed after another argument that we do not need to specify, the argument that stays as default would be signalized with plain brackets. Following the example of `'FillRect'`, if we wanted to specify the `rect` argument while leaving the `color` as default we would code it as follows:

```
>> Screen('FillRect', windowPtr, [], [100 100 600 600])
```

Opening window(s)

Opening the onscreen window, or windows, where our stimuli will appear is among the many things we can do with `Screen`. This is usually the first step using PTB, and the first PTB function that we use in an experimental script. PTB's help for `'OpenWindow'` shows the usage as follows:

```
>> help OpenWindow?

[windowPtr,rect]=Screen('OpenWindow',windowPtrOrScreenN
umber[,color][,rect][,pixelSize][,numberOfBuffers][,ste
reomode][,multisample][,imagingmode][,specialFlags][,cl
ientRect]);
```

However, most often the arguments we need in this case are:

```
% Opening Onscreen Window
```

```
>> [windowPtr, rect] = Screen('OpenWindow', 0, [127 127
127])
```

The `OpenWindow` command returns two output variables: `windowPtr` stands for Window's Pointer, which refers to the window we just opened, and `rect` stands for the dimensions of the monitor we are working with. The dimensions are signalized with a 1x4 vector with the following values [xMin yMin xMax yMax], the first two values will always be 0, the xMax will be the number of pixels for the x axe, i.e. the width of the screen, and the yMax will be the number of pixels for the y axe, i.e the height of the screen (see the Figure 1 below).

Figure 1. The dimensions of our monitor in pixels is the 1x4 vector variable `rect`. The first two values refer to the upper left corner, or the minimum values for the x and y axes, which are always 0,0; the third value refers to the upper right corner, or the maximum value for the x axe; and the fourth value refers to the lower left corner, or the maximum value for the y axe.

PTB default size for the onscreen window is full screen, and therefore the dimensions of our onscreen window will be the same as the dimensions of our monitor, as specified in the variable `rect` provided by `'OpenWindow'`. However we can change the exact dimensions and position of the onscreen window by writing a new `rect` (again as 1x4 vector with [xMin, yMin, xMax, yMax] dimensions) as an argument inside the `'OpenWindow'` command. The new `rect` will refer to the whole screen, e.g. `rect = [100 100 600 600]` means that the upper left corner of the onscreen window will be placed 100 pixels to the right on the x axe; 100 pixels down in the y axe, the right corner will be placed 600 pixels right in the x axe, and the right lower corner will be placed 600 pixels down in the y axe (see Figure 2 below). It is recommendable to use a smaller window to test a script as errors may occur, and the small screen is easier to close. On the contrary, if we are working with full screen and an error occurs throughout the script before closing the window, the only way we can close a full screen is by pressing Ctrl+Alt+Del and ending PTB.

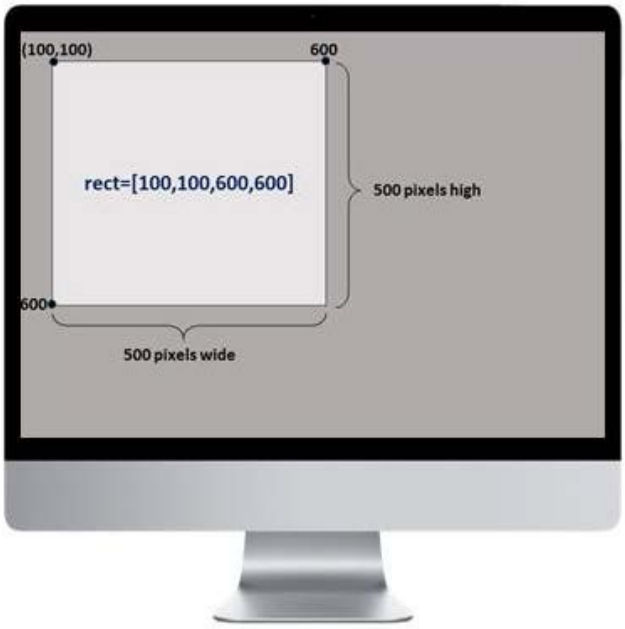

Figure 2. We can set a new `rect` for a smaller onscreen window. The dimensions of our new `rect` are local to the dimensions of our monitor. Therefore, for a `rect = [100 100 600 600]`, we will have an onscreen window of 500 high and 500 pixels wide.

Closing window(s)

PTB function `Screen` together with the command `'Close'` closes the window(s) we have opened. If we have more than one onscreen window open and we just want to close a specific one, we need to specify the `windowPtr` assigned to that window:

```
% Closing a specific onscreen window
>> Screen('Close', windowPtr)
```

In order to close all of them, even if it is just one, we use:

```
% Closing all onscreen windows
>> Screen('CloseAll')
```

or a shorter way to say the same:

```
>> sca
```

Screen 'Flip'

`Screen('Flip', windowPtr)` is a critical command line, as it is responsible for showing an onscreen window with the information we have specified in previous lines. When set as default, `Flip` shows all the settings that we have previously applied at the next screen vertical refresh. The usage allows further uses for more advanced actions such as specifying the exact time to flip, whether it should clear previous framebuffer, whether it should

synchronize with the refresh, or whether it should flip to just the onscreen identified by the windowPtr or to all on-screens in use. See specific examples of `Screen('Flip')` in use in the Writing or Drawing sections.

Show, hide, set mouse

Other PTB functions, non-directly pertaining to the stimulation, but useful and recommendable, are `HideCursor`, `ShowCursor` and `SetMouse`. The first and second functions respectively hide and show the mouse cursor on the screen. Most often they are placed just before opening and right after closing the onscreen window, so that the cursor does not interfere with our experiment and we can see the cursor after the experiment. `SetMouse` places the cursor on the screen coordinates of your choice.

```
% Hide the mouse on the screen
>> HideCursor

% Show the mouse on the screen
>> ShowCursor

% Place the cursor on x,y coordinates
>> SetMouse(x,y)
```

Writing

Writing is among the many edits that we can make in the onscreen window. This is helpful in a handful of situations such as writing the instructions at the beginning of the experiment or manipulating our stimuli if needed. `'DrawText'` is the command that will draw the text onto the screen. Different characteristics of the text can be manipulated with `Screen` commands and arguments. For instance, `'TextSize'`, `'TextFont'`, or `'TextStyle'`, are `Screen` commands that allow size, font or style changes in the writing (see below for usage). The `fontString` argument, placed within the `'TextFont'` command, refers to the name of the font you want to use (e.g. `'Helvetica'` or `'Times New Roman'`); and the argument `style`, within `'TextStyle'`, is coded with the following numbers: `0=normal`, `1=bold`, `2=italic`, `4=underline`, `8=outline`, `32=condense`, `64=extend`. Finally, `'DrawText'` contains the text you want to show, as well as its position on the onscreen (x,y coordinates) and its color. Color is specified as `[rgb]`, i.e. a `[red green blue]` 1x3 vector (e.g. `[255 0 0]` makes red, and `[255 255 0]` makes yellow).

Here below you will find the steps and code for writing text within the onscreen window, and a visual presentation in Figure 3 of how the execution of this code presents itself on the screen.

```
% Opening onscreen window
>> windowPtr=Screen('OpenWindow', 0, [], [100 100 1000
1000])

% Setting the text size
>> Screen('TextSize', windowPtr, 25);

% Setting the font
>> Screen('TextFont', windowPtr, 'Helvetica');

% Setting the style (underlined)
>> Screen('TextStyle', windowPtr, 4);

% Setting the text, location (x,y coordinates) and
color
>> Screen('DrawText', windowPtr, 'hey there', 300, 300,
[255 0 0])

% Showing the result on the screen
>> Screen('Flip', windowPtr)

% Keeping the window for 2 seconds
>> WaitSecs(2)

% Closing the window
>> sca
```

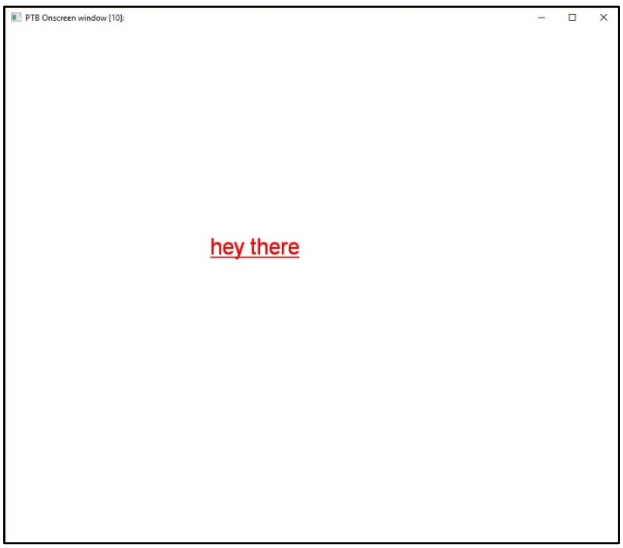

Figure 3. Result of executing the code written above. In this case, we opened an onscreen window with a `rect` of `[100 100 1000 1000]` dimensions and set up a text in red and underlined at the 300, 300 x,y pixel coordinates.

Drawing

PTB allows drawing lines and shapes. There is a specific `Screen('command')` for each shape, such as `'DrawLine'`, `'FillOval'`, `'FillRect'`, `'DrawDots'`, or `'DrawArc'` (type `Screen` in the MATLAB Command Window for a full list of subfunctions to draw). The arguments that go along with each command set the features of each shape, such as its color (same usage as in `'DrawText'`), position or size. For example, to draw a line (`'DrawLine'`) we need to set the x,y coordinates to where our line will start (fromH, fromH) and end (toH, toV), whereas `'FillRect'` will draw and fill a rectangle. In this case, `rect` specifies the dimension of the rectangle, with the same usage as `rect` in `'OpenWindow'`, that is a 1x4 vector with [xMin yMin xMax yMax] coordinates. If `rect` is set as default it will fill the whole onscreen window. You can see the usage of these two specific functions below.

```
% Drawing a line
>> Screen('DrawLine', windowPtr [,color], fromH, fromV,
toH, toV [,penWidth]);

% Drawing and filling a rectangle
>> Screen('FillRect', windowPtr [,color] [,rect] );
```

Showing images

Showing pictures in .jpg, .tiff or .png, among other formats, is a common need in Psychology experiments. PTB works with pictures in three steps: first the picture is embedded in a texture, then the texture is drawn and finally, the result is shown on the screen. Working with textures makes the script less time consuming, more efficient and avoids potential pixilation or artifacts on the screen. To create the texture, we need the command `'MakeTexture'` followed by the Windows Pointer and the image we want to show. The image needs to be in a matrix form, which means that it has already been read by MATLAB. See the example below with the necessary steps to read and show an image called 'MyImage.jpg' with PTB.

```
% Reading the picture
>> image = imread('MyImage.jpg');

% Creating the texture with the picture in it
>> texture = Screen('MakeTexture', windowPtr, image);

% Drawing the texture
>> Screen('DrawTexture', windowPtr, texture);

% It shows the result on the screen
>> Screen('Flip', windowPtr)
```

2.1.2 Auditory Stimulation

Playing sounds

The most accurate way to present auditory stimulation with PTB is using the PsychPortAudio sound driver, which is initialized by calling `InitializePsychSound`. In order to play a sound, we first need to read the sound file in MATLAB, e.g. by means of the `wavread` function. It is important to notice that PsychPortAudio interprets each row of the sound as a channel; therefore, we need to transpose the output of `wavread` before going further. Similar to the procedure we used to show images, playing sounds require three consecutive steps. First, we need to open a PsychPortAudio device. This is done by the command `'Open'`, whose output `pahandle` indicates the handle of the port audio device to be called later. Second, we fill the playback buffer of the audio device with the sound, using the command `'FillBuffer'`. In the third step, we actually play the buffered sound, by calling `'Start'`. Once the sound has been played, we need to `Close` the PsychPortAudio device. A code example is shown below:

```
% Initializing PsychPortAudio
>> InitializePsychSound

% Reading the sound
>> [mysound] = wavread('MySound.wav');
>> mysound = mysound';   % transposition

% Opening a PsychPortAudio device
>> pahandle = PsychPortAudio('Open')

% Filling buffer with sound
>> PsychPortAudio('FillBuffer', pahandle, mysound);

% Starting playback
>> PsychPortAudio('Start', pahandle);

% Closing the PsychPortAudio device
>> PsychPortAudio('Close', pahandle);
```

2.2 TIME

Stopping the script

Working with time is an essential part of stimulation. There are different ways to control how long something stays on the screen. One easy way is to stop the script for a given time. This can be useful when displaying instructions or pictures. `WaitSecs(TimeInSeconds)` is the function that stops the script for as long as it has

been set. This means that the onscreen window freezes with the last order that it was given in the script. Therefore, and continuing with the example used in the 'Showing images' section, we would add the `WaitSecs` order just after the `Flip` command to maintain the picture on the screen for the desired time (2 seconds in the example).

```
% Reading the picture
>> image = imread('MyImage.jpg');

% Creating the texture with the picture in it
>> texture = Screen('MakeTexture', windowPtr, image);

% Drawing the texture
>> Screen('DrawTexture', windowPtr, texture);

% It shows the result on the screen
>> Screen('Flip', windowPtr)

% Showing the picture for 2 seconds
>> WaitSecs(2)
```

Calculating Reaction Times

The behavioral data we record from the participant covers both RTs and responses. To compute the reaction times (RTs) we need to know how long it takes from a given point (usually the stimulus onset) to the time the participant responds, i.e. when a key is pressed. To know that interval of time we ask the computer for a time reference when the stimulus appears on the screen and again when a key is pressed, and then subtract the first to the latter to calculate the interval. `GetSecs` tells you how long the computer has been switched on in seconds with a wide range of decimals, which makes a very precise time reference. In short, the way we calculate RTs is:

```
% Calculating RTs
>> BeginningTrial = GetSecs;
BeginningTrial = 1.2509e+004

>> PressTime = GetSecs;
PressTime = 1.2510e+004;

>> RT = PressTime - BeginningTrial;
RT = 1.0731;
```

2.3 RESPONSE

Checking key presses

Responses are retrieved with `KbCheck`. This function gives very useful information on the response key and response time. The feedback that `KbCheck` gives is triple: `keyIsDown`, a true (1) or false (0) variable that indicates whether a key has been pressed; `secs`, the time reference to when the key was pressed (which is equivalent to retrieving `GetSecs` at the time the key was down), and `keyCode`, a 1x256 vector representing all the keys in the keyboard with a true (1) for the key that has been down and false (0) for the rest. To find out the correspondence of `keyCode` to the actual key that was pressed, we can use `KbName(find(keyCode))` to map the `keyCode` and have the name of key in return. If there are several devices connected via USB to the computer you can set the device/s you want the function to check. If `deviceNumber` is set up to -1, all keyboard devices will be checked, if it is set up to -2, all keypad devices will be checked, and if it is set up to -3 all keyboard and keypad devices will be checked. If `deviceNumber` is left blank, the main keyboard is checked.

```
% Checking which key has been pressed:
>> [keyIsDown, secs, keyCode] = KbCheck([deviceNumber])

% Finding the key that has been pressed
>> KeyPressed = KbName(find(keyCode));
```

Cleaning the event queue

It is important to make sure that the queue of events for key presses is empty. This means that we should make sure that `KbCheck` registers the key press form the participant as a response to the corresponding trial and not any other key press resulting from responding to other trial or an accidental press. Therefore, we recommend to call the function `FlushEvents` just before `KbCheck` to be sure that key press that we record for each trial corresponds to the response to that specific trial.

```
% Removing the queue of events for key presses
>> FlushEvents('keyDown')
```

Waiting for button press

There is also the possibility of asking PTB to stop the script, and therefore freeze the screen, until a key has been pressed. `KbWait` by itself does just that, it waits until a key has been pressed to continue reading the next lines in the script. To do that, it scans every 5 ms whether a key has been pressed. However, the function is more complete and can optionally give feedback on the key that has been pressed and give the temporal reference of the key press (just as `GetSecs` does). It is important to take into account that the temporal reference is the time the function checks whether a key has been pressed, and not the exact moment the key was pressed, so it is not recommendable to use it as a reliable RT measurement. Just as `KbCheck`, you can set this function to take into account specific devices (`deviceNumber`), or even a specific phase of the key press (e.g. pressing or releasing the key, by specifying different values of the argument `forWhat`).

```
% Wait for button press:
>> [secs, keyCode] = KbWait([deviceNumber] [, forWhat=0] [, untilTime=inf])
```

Checking mouse clicks

PTB can check for mouse clicks analogous to how it checks for key presses. The function `GetClicks` returns the number of clicks within the `interclickSecs` interval in the output variable `clicks`; it also returns where the cursor is positioned with x,y coordinates, and a 1xn vector (n=buttons in the mouse) with pressed (1) or not pressed (0) values in the column representing each button (`whichButton`).

```
% Get Mouse clicks:
>> [clicks, x, y, whichButton] = GetClicks ([windowPtrOrScreenNumber] [,interclickSecs] [,mouseDev])
```

3. PART 2: PRACTICE

In this section we will illustrate how to use Psychtoolbox to code a typical Psychology experiment from scratch. Programming this experiment will require using most of the PTB functions previously described, such as presentation of visual and auditory stimulation, time control and collection of responses using the keyboard. We recommend you take your time to understand how to program this task step by step. In this way, you will acquire the necessary skills to program your own (simpler or more complex) experiment. In the attached online material (see *Moodle Formación – almudena.capilla*) you can find the step-by-step MATLAB

scripts we will use in this section, as well as the stimuli needed to run them. In addition, the fragments of the scripts that need to be modified in each step will be highlighted in figures. Finally, you can find the final version of the PTB experiment in the Appendix section.

We propose you to program a simple categorization task, in which participants have to decide whether a given image contains an animal (press the keyboard with the right index finger) or not (left index finger response). We would like to collect responses for 10 trials, with the following structure each (see Figure 4 for a schematic illustration). First, a fixation cross will be shown for a variable interval between 0.4 and 1 seconds. This will be followed by the random presentation of an image (animal/non-animal) for a fixed duration of 0.5 seconds. Then, a question mark will appear on the screen until the participant gives a response or, in the case that no response is given, until a maximum interval of 2 seconds. Finally, we would like to give feedback for each trial, consisting of a 1000 Hz tone if the response was correct, and a 500 Hz tone in the case of an error.

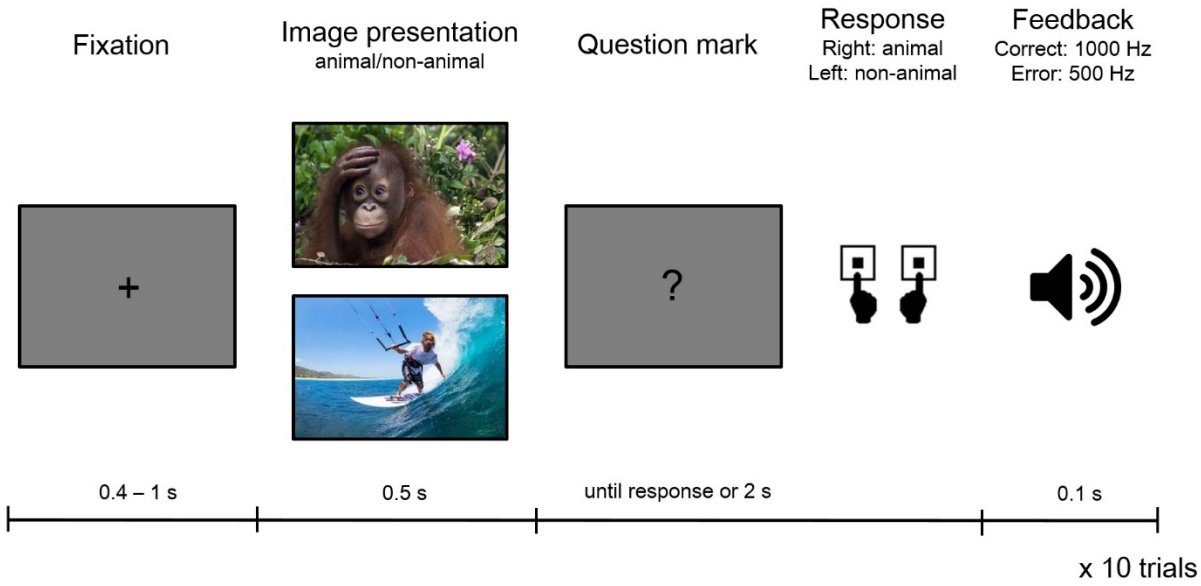

Figure 4. Schematic representation of the experimental task.

Now that we know what we would like to present to our participants, the first problem appears: where do we start? Shall we start showing images, randomizing them, collecting responses, or? Our recommendation is to start by writing down a prioritized list of steps you will need to sequentially achieve. Once you have prepared this list, you can start programming the experiment step by step. For this experimental task we have prepared the following steps, although you could try to develop your own list based on how you would build the experiment yourself.

Step 0: Opening (and closing) a window where the experiment will be presented
Step 1: Showing just one image
Step 2: Showing all animal images in a loop and a fixation cross before each of them

Step 3: Showing all images, saving information about the stimulus ID and condition of each image
Step 4: Collecting responses (key pressed and RT) and saving them in vectors
Step 5: Checking whether responses are correct, and saving this in a vector
Step 6: Giving auditory feedback (1000 Hz if correct response, 500 Hz if error)
Step 7: Randomizing stimulus/condition order and fixation time
Step 8: Showing instructions and a ready screen, and thanking the participant
Step 9: Converting the script into a function and including try-catch statement
Step 10: Saving experiment information and results in a file identified with participant's ID

Let's begin programming each step using PTB.

Step 0: Opening and closing an onscreen window

In this preparatory step, we are going to simply open the window where the experiment will be presented. For running the real experiment, we usually want to use full screen mode. However, as mentioned before, if an error occurs in this mode, you will need to end PTB (and MATLAB) by pressing Ctrl+Alt+Del. Thus, while you are debugging the experiment, we recommend you to open a smaller window. In the case of an error, you could always access the Command Window to close the screen manually by writing `Screen('CloseAll')`or, alternatively, `sca`. In the script Step 0, you could choose between full screen and small window by un/commenting the line of code of one of them, as it is shown in the figure below.

If you run the script Step0 you will see that it simply opens and closes a gray on-screen window. If you want to keep it open longer (e.g. 2 seconds) you could add `WaitSecs(2)` between the open and close commands.

Additionally, we suggest you to run the command `VBLSyncTest` to test the synchronization accuracy of PTB with the vertical retrace on your computer. Once you have checked that synchronization is correct, you can comment that line in your script so that this is not executed every time you run the experiment.

```
1 -   screenNumber = 0;
2
3 -   VBLSyncTest          % synchronization test before starting
4
5     % Choose the size of the window to open:
6     % --- 1- Full screen presentation: for the actual experiment
7     % rect = [];
8
9     % --- 2- Small window presentation: for debugging
10 -  rect = [600 100 1300 700];
11
12    % Opening the window
13 -  [windowPtr,rect] = Screen('OpenWindow',screenNumber,[127 127 127],rect);
14
15    % Closing the window
16 -  Screen('CloseAll');  % you can also type sca for closing the window
17
```

Step 1: Showing one image

The aim of Step 1 is to present just one image (e.g. `'NA5.jpg'`) in the on-screen window. In order to do so, we first need to move to the folder where the image is stored on your computer (`cd`) and read it using the MATLAB function `imread`. Then, the image will be converted into a texture. Textures can be presented fast and very precisely by PTB. As we showed you before, this is done using three consecutive `Screen` commands: (1) `'MakeTexture'` to convert the image into a texture, (2) `'DrawTexture'` to draw the specified texture and, (3) `'Flip'` to present the Texture synchronized with the next vertical retrace.

Once the image is shown on your screen you could hold it until a key is pressed using `KbWait`. After that, we propose you to show a gray window for one second before closing the onscreen window.

```
14      %%  ---------  Showing one jpg image  ---------  %%
15
16      % Change directory to the images path
17 -    dpath='C:\Psychtoolbox';
18 -    addpath (dpath)
19 -    cd([dpath '\Stim\Images'])
20
21      % Read the image (img will be a 3D matrix: height x width x RGB color)
22 -    img = imread('NA5.jpg');
23
24 -    texture = Screen('MakeTexture',windowPtr,img);
25 -    Screen ('DrawTexture',windowPtr,texture);
26 -    Screen ('Flip',windowPtr);   % this is the exact moment when the image is shown
27
28      % Hold the image until a key is pressed
29 -    KbWait
30
31
32      %% Background window (gray = 127) (black 0; white 255)
33 -    Screen ('FillRect',windowPtr,127);
34 -    Screen ('Flip',windowPtr);
35 -    WaitSecs (1)
36
```

Step 2: Showing a fixation cross and all animal images in a loop

In this step we are going to read and present all the animal images for 1.5 seconds and a fixation cross before them for 0.5 seconds. This timing is for debugging; we will later change it to the real one. As the animal images are consecutively named from `'A1.jpg'` to `'A5.jpg'`, we can read them using a `for` loop, and store them in a variable called `animal` consisting of a 1x5 cell matrix for later use. Alternatively, you could also automatically read all the images in a folder using `ls('*.jpg')`. This is especially useful if your images do not have systematic names.

After reading the images and the fixation cross, you can show the fixation cross followed by each image in consecutive order using another `for` loop. As already practiced in Step 1, the presentation of any image will require three commands: `'MakeTexture'`, `'DrawTexture'` and `'Flip'`.

You may wonder why we do not take the same `for` loop for reading the images and presenting the textures. We recommend programming two separate `for` loops because reading images is time consuming and if it was to be included in the presentation loop, it would damage presentation time accuracy. You could check how much time is spent on a selected piece of code by using the MATLAB functions `tic` and `toc`.

```
%%  ---------  Showing all the animal images  ---------  %%
% Images will be shown for 1.5 s
% Before presenting each image, a fixation point will be shown for 0.5 s

% Change directory to the images path
dpath='C:\Psychtoolbox';
addpath (dpath)
cd([dpath '\Stim\Images'])

% Read the 5 animal images
Nanimal = 5;
for i = 1:Nanimal
    animal{i} = imread(['A' num2str(i) '.jpg']);
end

% Change directory to the folder where the fixation point is
cd([dpath '\Stim\'])

% Read the fixation point image
fixation = imread('Fixation.jpg');

for i = 1:Nanimal

    % Fixation point shown for 0.5 s
    texture = Screen('MakeTexture',windowPtr,fixation);     % fixation point
    Screen ('DrawTexture',windowPtr,texture);
    Screen ('Flip',windowPtr);
    WaitSecs (0.5)

    % Image shown for 1.5 s
    texture = Screen('MakeTexture',windowPtr,animal{i});    % each animal
    Screen ('DrawTexture',windowPtr,texture);
    Screen ('Flip',windowPtr);
    WaitSecs (1.5)

end

```

Step 3: Showing all images, saving information about stimulus ID and condition

In Step 3 we aim to show all images in order (first, animals; second, non-animal images). Importantly, we would like to store information about which stimulus is presented in each trial (both its ID and condition). This is not really relevant yet, as stimuli are presented in a known sequence, but it will become essential in next steps, when the stimulus order is randomized.

For this example, we have manually created the stimulus ID and condition vectors. However, you could create them more automatically, which is useful if you have a large number of stimuli, e.g. `condition = [ones(Nanimal,1); 2*ones(Nnoanimal,1)]`; another solution could be reading this information from a file (e.g. from an excel file, using `xlsread`).

As you can see in the figure below, in the trial presentation loop we have used the conditional statement `if` to select whether the trial is presenting an animal (if condition for this trial is 1) or a non-animal (if condition is 2). Apart from that, the code for presenting the fixation cross and the animal/non-animal picture in each trial is exactly the same as in Step 2.

```
%%  -----  Showing all images, saving information about stimulus ID and condition of each image  -----  %%
% Images will be shown for 1.5 s
% Before presenting each image, a fixation point will be shown for 0.5 s

% Vector stimulus contains information about the ID of each image
% Vector condition contains information about image category (animal = 1; non-animal = 2)

% Change directory to the images path
dpath='C:\Psychtoolbox';
addpath (dpath)
cd([dpath '\Stim\Images'])

% Read all the images (5 animals + 5 non-animals)
Nanimal = 5;
Nnonanimal = 5;
Ntrials = Nanimal + Nnonanimal;

for i = 1:Nanimal
    animal{i} = imread(['A' num2str(i) '.jpg']);
end

for i = 1:Nnonanimal
    non_animal{i} = imread(['NA' num2str(i) '.jpg']);
end

% Create vectors stimulus and condition for all the trials (Ntrials = 10)
stimulus = [1 2 3 4 5 1 2 3 4 5]';
condition = [1 1 1 1 1 2 2 2 2 2]';
```

```
for i = 1:Ntrials

    % Show fixation point for 0.5 s
    texture = Screen('MakeTexture',windowPtr,fixation);
    Screen ('DrawTexture',windowPtr,texture);
    Screen ('Flip',windowPtr);
    WaitSecs (0.5)

    % Show images for 1.5 s
    % First show all animals; then show non animals (as the order of the vector condition indicates)
    % Show images from 1 to 5 (as the vector stimulus indicates)

    if condition(i) == 1          % condition for trial i is animal
        stimulus_id = stimulus(i);
        texture = Screen('MakeTexture',windowPtr,animal{stimulus_id});
    elseif condition(i) == 2      % condition for trial i is non-animal
        stimulus_id = stimulus(i);
        texture = Screen('MakeTexture',windowPtr,non_animal{stimulus_id});
    end
    Screen ('DrawTexture',windowPtr,texture);
    Screen ('Flip',windowPtr);
    WaitSecs (1.5)

end
```

Step 4: Collecting responses (key pressed and RT) and saving them in vectors

The aim of this step is to collect the participant's responses in each trial. In order to do so, after each image presentation, we need to include a piece of code to check for responses. One easy way to do this would be to use `[t2, keyCode] = KbWait([], [], 2)`, which would check whether a key has been pressed during 2 seconds. However, in Step 4 we alternatively suggest you to use `KbCheck` inside a `while` loop running for 2 seconds. Although the former alternative is simpler, the latter is more precise and can be more useful in the case of more complex scenarios.

In each iteration of the `while` loop, PTB will check whether a key has been pressed. As already said, it is important to `FlushEvents` beforehand to avoid responses from previous trials to be erroneously detected. In the case that a key press is detected, we will ask PTB to find its identity by using `KbName`. This will be stored in a 1xNtrials cell variable called `response_key`. In addition, we are interested in collecting the RTs in the variable `response_time`. To obtain RTs, we will take a time reference with `GetSecs` before starting the `while` loop (`t1`) and calculate how much time has elapsed until a key has been pressed (`t2`). If a key press has been detected before the time limit of 2 seconds, the `while` loop will `break`, and the experiment will continue.

Finally, it is a good practice to initialize the response vectors before filling them with experimental data. We can create vectors for storing the key pressed, RTs (and correct responses, which will be used in the next step) with the same length as the total number of trials, and filled them with `NaNs` (Not a Number, i.e. missing data). In this way, if a participant does not provide response for a given trial, this will automatically remain as a missing value.

```
39      % Create  vectors to save subject's responses: key pressed, correct/error and RT for each trial
40 -    response_key = cell (Ntrials,1);
41 -    response_correct = NaN (Ntrials,1);
42 -    response_time = NaN (Ntrials,1);
43
```

```
84
85 -        FlushEvents('keyDown');
86 -        t1 = GetSecs;
87 -        time = 0;
88 -        while time < 2          % maximum wait time: 2 s
89 -            [keyIsDown,t2,keyCode] = KbCheck;    % determine state of keyboard
90 -            time = t2-t1;
91
92 -            if (keyIsDown)      % has a key been pressed?
93 -                key = KbName(find(keyCode));            % find key's name
94 -                response_key{i} = key;
95 -                response_time(i) = time;
96 -                break;
97 -            end
98 -        end
99
100 -   end
101
```

Step 5: Checking whether responses are correct, and saving them in a vector

In addition to recording the key pressed and RT for a given trial (Step 4), in this step we aim to evaluate whether this response is correct. If the image shown is an animal and the subject pressed the `'k'` key (right hand) the response will be correct. It will also be correct if the image was a non-animal and the key pressed was `'d'` (left hand). This information will be stored in the vector `response_correct`, which will take values equal to 1 for correct responses and to 0 for errors.

```
84
85 -        FlushEvents('keyDown');
86 -        t1 = GetSecs;
87 -        time = 0;
88 -        while time < 2          % maximum wait time: 2 s
89 -            [keyIsDown,t2,keyCode] = KbCheck;    % determine state of keyboard
90 -            time = t2-t1;
91
92 -            if (keyIsDown)      % has a key been pressed?
93 -                key = KbName(find(keyCode));            % find key's name
94 -                response_key{i} = key;
95 -                response_time(i) = time;
96
97 -                if  condition(i) == 1   &&  strcmp(response_key{i},'k') == 1     % animal and right key (k)
98 -                    response_correct(i) = 1;        % it's correct
99 -                elseif condition(i) == 1   &&  strcmp(response_key{i},'d') == 1  % animal and left key (d)
100 -                   response_correct(i) = 0;        % it's an error
101 -               elseif condition(i) == 2   &&  strcmp(response_key{i},'k') == 1  % non-animal and right key (k)
102 -                   response_correct(i) = 0;        % it's an error
103 -               elseif condition(i) == 2   &&  strcmp(response_key{i},'d') == 1  % non-animal and left key (d)
104 -                   response_correct(i) = 1;        % it's correct
105 -               end
106
107 -               break;
108
109 -           end
110
111 -       end
112
113 -   end
114
```

Step 6: Giving auditory feedback (1000 Hz if correct response, 500 Hz if error)

Once we know whether the response to a given trial is correct, we can deliver the feedback for this trial by means of auditory stimulation (a 1000 Hz tone if the response was correct, and a 500 Hz tone if it was an error). As indicated in the introductory section, the most accurate way to present auditory stimulation is by means of the PsychPortAudio sound driver. After reading the sound file in MATLAB, we can play it by calling three consecutive `PsychPortAudio` commands: `'Open'` the port audio device, `'FillBuffer'` with sound and `'Start'` playback. After using the PsychPortAudio device, we should `Close` it.

```
52
53      % Change directory to the folder where the sounds are
54 -    cd([dpath '\Stim\Sounds'])
55
56      % Read the sounds
57 -    sound_correct = wavread('Sound_1000Hz');
58 -    sound_correct = sound_correct';
59
60 -    sound_error = wavread('Sound_500Hz');
61 -    sound_error = sound_error';
62
63      % Initializing and opening a PsychPortAudio device
64 -    InitializePsychSound
65 -    pahandle_correct = PsychPortAudio('Open');
66 -    pahandle_error = PsychPortAudio('Open');
67
68      % Filling buffer with sound
69 -    PsychPortAudio('FillBuffer', pahandle_correct, sound_correct);
70 -    PsychPortAudio('FillBuffer', pahandle_error, sound_error);
71
```

```
109
110 -           if (keyIsDown)      % has a key been pressed?
111 -               key = KbName(find(keyCode));            % find key's name
112 -               response_key{i} = key;
113 -               response_time(i) = time;
114
115 -               if  condition(i) == 1   &&  strcmp(response_key{i},'k') == 1    % animal and right key (k)
116 -                   response_correct(i) = 1;        % it's correct
117 -                   PsychPortAudio('Start', pahandle_correct);
118 -               elseif condition(i) == 1   &&  strcmp(response_key{i},'d') == 1 % animal and left key (d)
119 -                   response_correct(i) = 0;        % it's an error
120 -                   PsychPortAudio('Start', pahandle_error);
121 -               elseif condition(i) == 2   &&  strcmp(response_key{i},'k') == 1 % non-animal and right key (k)
122 -                   response_correct(i) = 0;        % it's an error
123 -                   PsychPortAudio('Start', pahandle_error);
124 -               elseif condition(i) == 2   &&  strcmp(response_key{i},'d') == 1 % non-animal and left key (d)
125 -                   response_correct(i) = 1;        % it's correct
126 -                   PsychPortAudio('Start', pahandle_correct);
127 -               end
128
129 -               break;
130
131 -           end
132
133 -       end
134
135 -   end
136
```

```
143
144     %% Closing the PsychPortAudio device
145 -   PsychPortAudio('Close', pahandle_correct);
146 -   PsychPortAudio('Close', pahandle_error);
147
```

Step 7: Randomizing stimulus/condition order and fixation time

One critical issue in the design of experiments in Psychology is to control for confounding variables. One efficient way to accomplish this is to create a random sequence of stimuli for each participant. In our case, we could first shuffle trial order by using `randperm`, and then we could apply this random order to stimulus ID and condition. Additionally, we might also want to randomize timing. In our case, the time of image presentation and auditory stimulation was planned to be fixed. On the contrary, we planned to have a variable interval for fixation, from 0.4 to 1 second. This can be achieved using the MATLAB `rand` function.

Finally, it is very important to bear in mind that the functions employed to randomize events (`rand` and `randperm`) depend on the random number generator of MATLAB. Before calling them it is critical to seed this generator to a random state (e.g. based on current time: `seed = round(sum(10*clock))`, and setting the random number generator state to this number by `rng(seed)`. It is a good practice to save the seed value together with the experimental session information. In this way we could replicate the stimulation protocol for each participant in case we need it.

```
71
72      %% Randomization of stimulus/condition order and fixation time
73
74 -    seed = round(sum(10*clock));           % set a "random" seed for this experimental session
75 -    rng(seed)
76
77 -    random_order = randperm(Ntrials);
78 -    stimulus = stimulus (random_order);      % assign random order to stimuli
79 -    condition = condition (random_order);
80
81 -    fixation_time = rand(Ntrials,1)*(1-0.4) + 0.4;   % random number between 0.4 and 1
82
```

Step 8: Showing instructions and a ready screen, and thanking the participant

Once the main experiment is ready and we have checked that it runs as planned, it is time for adding some supplementary elements, such as instructions, a training session before starting the experiment, or a final screen to thank the participant at the end. In this example, we suggest you to add the instructions, a 'ready' and a 'thanks' screen. There are two ways of presenting text on the screen: by using `'DrawText'` or by presenting an image that we have previously created with the text we need to show. In this experiment we are going to use both ways for the sake of illustration. In this case, we read the images for the instructions and the ready screen and present them as textures as described in Step 2, and thank the participant with the `'DrawText'` function.

When thanking the participant, we would like the text to appear in the center of the screen. To calculate the coordinates to place the text in the center, we have first calculated the x,y coordinates for the middle of the screen and then set up the characteristics of the text. The `Screen` command `'TextBounds'` (`textRect = Screen('TextBounds', windowPtr, text)` returns the `rect` of the text. Next, we have subtracted half of the width and height of the text to calculate the x,y coordinates where our text should begin. Finally, we just need to draw and flip the text.

```
53      % Read images for instructions  and  ready
54 -    instructions1 = imread('Instructions_1.jpg');
55 -    instructions2 = imread('Instructions_2.jpg');
56 -    ready = imread('Ready.jpg');
57 -
```

```
90      %% Showing instructions and a ready screen before starting the experiment
91
92 -    texture = Screen('MakeTexture',windowPtr,instructions1);
93 -    Screen ('DrawTexture',windowPtr,texture);
94 -    Screen ('Flip',windowPtr);
95 -    WaitSecs(1)
96 -    KbWait
97
98 -    texture = Screen('MakeTexture',windowPtr,instructions2);
99 -    Screen ('DrawTexture',windowPtr,texture);
100 -   Screen ('Flip',windowPtr);
101 -   WaitSecs(1)
102 -   KbWait
103
104 -   texture = Screen('MakeTexture',windowPtr,ready);
105 -   Screen ('DrawTexture',windowPtr,texture);
106 -   Screen ('Flip',windowPtr);
107 -   WaitSecs(1)
108 -   KbWait
109
```

```
181     %% Thanking the participant
182 -   text='Thanks!';
183
184     % Characteristics of the text
185 -   Screen('TextFont',windowPtr,'Helvetica');
186 -   Screen('TextSize',windowPtr,52);
187
188     % Center of the screen
189 -   centerX = rect(3)/2;
190 -   centerY = rect(4)/2;
191
192     % Text size in pixels
193 -   textRect = Screen('TextBounds',windowPtr,text);
194 -   textWidth = textRect(3);
195 -   textHeight = textRect(4);
196
197     % Text coordinates
198 -   textX = centerX - textWidth/2;
199 -   textY = centerY - textHeight/2;
200
201     % Drawing text
202 -   Screen('DrawText',windowPtr,text,textX,textY,[0 0 0]);
203 -   Screen('Flip', windowPtr);
204
205 -   WaitSecs(3)
206
```

Step 9: Converting the script into a function and including try-catch statement

In order to easily call the experiment from the Command Window, we need to convert the script into a function by adding the term `function` in the first line of our script. The name of the .m file should be the same as the one we assigned to the function. In this case, this is Step_9 (please, remember not to leave blank spaces between words in MATLAB; you can use _ instead). However, we recommend to change it by a more meaningful name such as AnimalCategorization, as we do in the final script. The right-hand arguments of the function are input variables, for example, the `subject` number that will be used to save the experiment information for each participant. The left-hand arguments of the function are

output variables, which in our case will be a variable containing the `results` of the experiment, as we will explain in detail in next step.

In addition, it is a good practice to include a try-catch statement. Thus, PTB would try to run the experiment, which is placed in the `try` block. In case it encountered an error, the `catch` block would be executed. In this way we ensure that onscreen windows get close and that we get access to the Command Window again. Otherwise, we would need to end PTB by pressing Ctrl+Alt+Del if any error occurred during the experiment. We can also use `psychrethrow(psychlasterror)` to ask PTB for the last error that made the experiment crash.

```
1   function [results] = Step_9 (subject)
2
3 -  try
4
220
221 -  catch      % this catch section closes the window automatically in case of an error
222
223 -      Screen('CloseAll');
224 -      psychrethrow(psychlasterror);     % throw last error
225
226 -  end
227
```

Step 10: Saving experiment information and results in a file identified with participant's ID

As a final step we want to save all the information generated during the experiment in a .mat file identified with the participant's code number (`'AnimalCategorization_SubjectX.mat'`; X is subject number). A well-organized way to save this information is as a MATLAB structure. Thus, we would have a variable called `results`, with several fields, e.g. `subject` number, `seed` of the random number generator, `stimulus` or `response`. In turn, some of these fields might be comprised by subsequent subfields. For instance, the field `stimulus` contains all the relevant information regarding the stimulation employed.

```
1    function [results] = Step_10 (subject)
2
```

```
212      %% Saving experiment information and results in a file called AnimalCategorization_SubjectX.mat
213 -    results.subject = subject;
214 -    results.seed    = seed;
215 -    results.ntrials = Ntrials;
216
217 -    results.stimulus.id = stimulus;
218 -    results.stimulus.condition = condition;
219 -    results.stimulus.stim_time = 0.5;
220 -    results.stimulus.fixation_time = fixation_time;
221
222 -    results.response.key = response_key;
223 -    results.response.correct = response_correct;
224 -    results.response.reaction_time = response_time;
225
226 -    cd([dpath '\Stim\Results'])
227 -    fout=sprintf('AnimalCategorization_Subject%d.mat', subject);
228 -    save(fout, 'results');
229
```

Now your experiment is ready! You can change the name of the function by AnimalCategorization and run your first participant by writing in the Command Window:

```
[results] = AnimalCategorization (1)
```

APPENDIX

```
function [results] = AnimalCategorization (subject)

try

    screenNumber = 0;

    % Choose the size of the window to open:
    % --- 1- Full screen presentation: for the actual experiment
    % rect = [];

    % --- 2- Small window presentation: for debugging
    rect = [600 100 1300 700];

    % Opening the window
    [windowPtr,rect] = Screen('OpenWindow',screenNumber,[127 127 127],rect);


    %%  -----  Reading all necessary files and creating stimulus and response vectors  -----
%%

    % Change directory to the images path
    dpath='C:\Psychtoolbox';
    addpath (dpath)
    cd([dpath '\Stim\Images'])

    % Read all the images (5 animals + 5 non-animals)
    Nanimal = 5;
    Nnonanimal = 5;
    Ntrials = Nanimal + Nnonanimal;

    for i = 1:Nanimal
        animal{i} = imread(['A' num2str(i) '.jpg']);
    end

    for i = 1:Nnonanimal
        non_animal{i} = imread(['NA' num2str(i) '.jpg']);
    end

    % Vector stimulus contains information about the ID of each image
    % Vector condition contains information about image category (animal = 1; non-animal = 2)
    stimulus = [1 2 3 4 5 1 2 3 4 5]';
    condition = [1 1 1 1 1 2 2 2 2 2]';

    % Create  vectors to save subject's responses: key pressed, correct/error and RT for each
trial
    response_key = cell (Ntrials,1);
    response_correct = NaN (Ntrials,1);
    response_time = NaN (Ntrials,1);

    % Change directory to the folder where the fixation point is
    cd([dpath '\Stim\'])

    % Read the fixation point image
    fixation = imread('Fixation.jpg');

    % Read the image with the question mark
    question = imread('Question.jpg');

    % Read images for instructions and ready
    instructions1 = imread('Instructions_1.jpg');
    instructions2 = imread('Instructions_2.jpg');
    ready = imread('Ready.jpg');


    % Change directory to the folder where the sounds are
    cd([dpath '\Stim\Sounds'])

    % Read the sounds
```

```
    sound_correct = wavread('Sound_1000Hz');
    sound_correct = sound_correct';

    sound_error = wavread('Sound_500Hz');
    sound_error = sound_error';

    % Initializing and opening a PsychPortAudio device
    InitializePsychSound
    pahandle_correct = PsychPortAudio('Open');
    pahandle_error = PsychPortAudio('Open');

    % Filling buffer with sound
    PsychPortAudio('FillBuffer', pahandle_correct, sound_correct);
    PsychPortAudio('FillBuffer', pahandle_error, sound_error);

    %% Randomization of stimulus/condition order and fixation time

    seed = round(sum(10*clock));           % set a "random" seed for this experimental session
    rng(seed)

    random_order = randperm(Ntrials);
    stimulus = stimulus (random_order);      % assign random order to stimuli
    condition = condition (random_order);

    fixation_time = rand(Ntrials,1)*(1-0.4) + 0.4;   % random number between 0.4 and 1


    %% Showing instructions and a ready screen before starting the experiment

    texture = Screen('MakeTexture',windowPtr,instructions1);
    Screen ('DrawTexture',windowPtr,texture);
    Screen ('Flip',windowPtr);
    WaitSecs(1)
    KbWait

    texture = Screen('MakeTexture',windowPtr,instructions2);
    Screen ('DrawTexture',windowPtr,texture);
    Screen ('Flip',windowPtr);
    WaitSecs(1)
    KbWait

    texture = Screen('MakeTexture',windowPtr,ready);
    Screen ('DrawTexture',windowPtr,texture);
    Screen ('Flip',windowPtr);
    WaitSecs(1)
    KbWait


    %% ----- Checking whether responses are correct and giving auditory feedback (1000Hz
correct, 500 Hz error) ----- %%

    for i = 1:Ntrials

        % Show fixation point for a random duration (between 0.4 and 1 s) as stored in the
vector fixation_time
        texture = Screen('MakeTexture',windowPtr,fixation);
        Screen ('DrawTexture',windowPtr,texture);
        Screen ('Flip',windowPtr);
        WaitSecs (fixation_time(i))

        % Show images for 0.5 s
        % Show images following the random order of vectors stimulus
        if condition(i) == 1          % animal
            stimulus_id = stimulus(i);
            texture = Screen('MakeTexture',windowPtr,animal{stimulus_id});
        elseif condition(i) == 2      % non-animal
            stimulus_id = stimulus(i);
            texture = Screen('MakeTexture',windowPtr,non_animal{stimulus_id});
        end
        Screen ('DrawTexture',windowPtr,texture);
        Screen ('Flip',windowPtr);
        WaitSecs (0.5)
```

```
        % Show question mark until the subject gives a response
        % If there is no response in 2 s, start next trial
        texture = Screen('MakeTexture',windowPtr,question);
        Screen ('DrawTexture',windowPtr,texture);
        Screen ('Flip',windowPtr);

        FlushEvents('keyDown');
        t1 = GetSecs;
        time = 0;
        while time < 2         % maximum wait time: 2 s
            [keyIsDown,t2,keyCode] = KbCheck;    % determine state of keyboard
            time = t2-t1;

            if (keyIsDown)      % has a key been pressed?
                key = KbName(find(keyCode));            % find key's name
                response_key{i} = key;
                response_time(i) = time;

                if  condition(i) == 1   &&  strcmp(response_key{i},'k') == 1    % animal &
right
                    response_correct(i) = 1;        % it's correct
                    PsychPortAudio('Start', pahandle_correct);
                elseif condition(i) == 1   &&  strcmp(response_key{i},'d') == 1 % animal &left
                    response_correct(i) = 0;        % it's an error
                    PsychPortAudio('Start', pahandle_error);
                elseif condition(i) == 2   &&  strcmp(response_key{i},'k') == 1 % n-animal
&right
                    response_correct(i) = 0;        % it's an error
                    PsychPortAudio('Start', pahandle_error);
                elseif condition(i) == 2   &&  strcmp(response_key{i},'d') == 1 % n-animal
&left
                    response_correct(i) = 1;        % it's correct
                    PsychPortAudio('Start', pahandle_correct);
                end

                break;

            end

        end

    end


    %% Background window (gray = 127) (black 0; white 255)
    Screen ('FillRect',windowPtr,127);
    Screen ('Flip',windowPtr);
    WaitSecs (1)


    %% Thanking the participant
    text='Thanks!';

    % Characteristics of the text
    Screen('TextFont',windowPtr,'Helvetica');
    Screen('TextSize',windowPtr,52);

    % Center of the screen
    centerX = rect(3)/2;
    centerY = rect(4)/2;

    % Text size in pixels
    textRect = Screen('TextBounds',windowPtr,text);
    textWidth = textRect(3);
    textHeight = textRect(4);

    % Text coordinates
    textX = centerX - textWidth/2;
    textY = centerY - textHeight/2;

    % Drawing text
```

```
    Screen('DrawText',windowPtr,text,textX,textY,[0 0 0]);
    Screen('Flip', windowPtr);

    WaitSecs(3)


    %% Saving experiment information and results in a file called
AnimalCategorization_SubjectX.mat (X is subject number)
    results.subject = subject;
    results.seed    = seed;
    results.ntrials = Ntrials;

    results.stimulus.id = stimulus;
    results.stimulus.condition = condition;
    results.stimulus.stim_time = 0.5;
    results.stimulus.fixation_time = fixation_time;

    results.response.key = response_key;
    results.response.correct = response_correct;
    results.response.reaction_time = response_time;

    cd([dpath '\Results'])
    fout=sprintf('AnimalCategorization_Subject%d.mat', subject);
    save(fout, 'results');


    %% Closing the PsychPortAudio device
    PsychPortAudio('Close', pahandle_correct);
    PsychPortAudio('Close', pahandle_error);


    %% Closing the window
    Screen('CloseAll');  % sca


catch      % this catch section closes the window automatically in case of an error

    Screen('CloseAll');
    psychrethrow(psychlasterror);     % throw last error

end
```