o_ptb

Thomas Hartmann

Apr 26, 2023

CONTENTS

1 Introduction **1**

2 How to cite the o_ptb **3**

2.1 How to install o_ptb 3

2.2 How to use the o_ptb. A step by step tutorial 4

2.3 Reference 38

3 Indices and tables **63**

MATLAB Module Index **65**

Index **67**

CHAPTER

ONE

INTRODUCTION

o_ptb is a class based library that runs on top of the well-known Psychtoolbox. It also uses the VPixx system when connected. The latest version also include support for the LabJack U3 device for triggering.

The library tries to achieve two goals:

1. The same code should run and behave as equally as possible whether the Vpixx system is connected or not. If it is not connected, its capabilities will be emulated using the computer's hardware.
2. Make handling visual and auditory stimuli as well as triggers and responses easier and more coherent.

CHAPTER

TWO

HOW TO CITE THE O_PTB

If you have used the o_ptb, you would do us and the authors of the Psychtoolbox a great favor if you cited the following articles:

For the o_ptb please cite:

- Hartmann, T & Weisz, N. (2020). An Introduction to the Objective Psychophysics Toolbox (o_ptb), Front. Psychol. 11:585437

For the Psychtoolbox please refer their website: http://psychtoolbox.org/credits. Currently, they list the following articles they would like you to cite:

- Brainard, D. H. (1997) The Psychophysics Toolbox, Spatial Vision 10:433-436
- Pelli, D. G. (1997) The VideoToolbox software for visual psychophysics: Transforming numbers into movies, Spatial Vision 10:437-442
- Kleiner M, Brainard D, Pelli D, 2007 What's new in Psychtoolbox-3?, Perception 36 ECVP Abstract Supplement

2.1 How to install o_ptb

2.1.1 Requirements

In order to develop and run experiments with o_ptb, you need to have:

1. Matlab 2019b or higher
2. The signal processing toolbox of Matlab
3. A current version of the Psychtoolbox

If you want to use a Labjack device to trigger, you also need a recent version of python.

2.1.2 Download o_ptb

The preferred way to download o_ptb is via git. This enables you to easily update o_ptb to the most current version. You can also download it directly from the gitlab repository as a zip file, but this method is not recommended.

Assuming, you have git installed on your computer, you can download o_ptb by "cloning" the repository.

Open a terminal and navigate to the folder under which you want to find o_ptb.

Let's suppose, I have a folder called `matlab_toolboxes` where I store all my toolboxes for Matlab:

```
cd matlab_toolboxes
git clone https://gitlab.com/thht/o_ptb.git
```

This should be rather quick to finish. You should now see a new folder called `o_ptb`.

2.1.3 Keep o_ptb up-to-date

o_ptb is under constant development. Make sure that you always keep your version updated. In order to do so, open a terminal, navigate to the folder in which o_ptb lives and issue `pull` command of git:

```
cd matlab_toolboxes/o_ptb
git pull
```

2.1.4 Where to go next

Now that you have successfully installed o_ptb, you might want to take a look at the *tutorials*.

2.2 How to use the o_ptb. A step by step tutorial

In this tutorial, you will learn how to use o_ptb.

In order to use it efficiently, you also need to learn a few "advanced" programming techniques provided by Matlab. Don't be scared by the word "advanced"! Don't think of it as "super complex" but rather as a way to achieve awesome stuff.

This tutorial this consists of two parts. The first describes the Matlab features required. The second describes the features of the o_ptb.

2.2.1 About packages and classes

About Packages

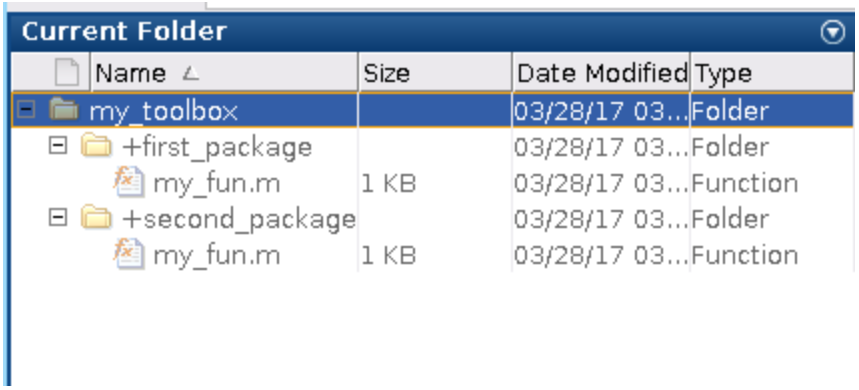
Did you ever think that the Path system of Matlab is not that good? Do you think, it is annoying that you always need to add a prefix (like `ft_` or `obob_`) to your functions because otherwise there might be more than one function sharing the same name which creates chaos?

Packages are here to help you!

What is a "Package"?

A Package is a folder that start with a +. Everything coming afterwards is the name of the package. Function you put inside of a folder starting with a + are part of that package.

Take a look here:



You see a folder called `my_toolbox`. This could be a parent folder for a new toolbox you are writing. In order to use your new toolbox, people are supposed to **only** add this parent folder.

Inside, you find two packages: `first_package` and `second_package`. Each of these packages has a function called `my_fun.m`.

You can see that both functions have the same name. This would not work normally. The trick here is that both functions are in separate packages!

How do I call a function in a package?

Packages are available as soon as the *parent folder* of the package is in the Matlab path. So, in our case, we just need to add `my_toolbox` to the Matlab path in order to be able to access both packages:

```
restoredefaultpath
addpath('my_toolbox');
```

Now we can call our functions like this:

```
first_package.my_fun();
second_package.my_fun();
```

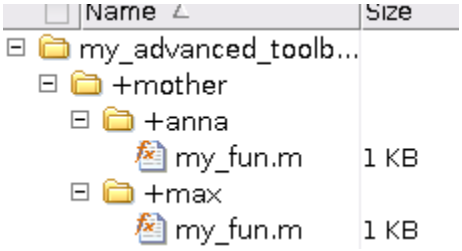
So, we write the name of the package first, the put a dot (`.`) and then the name of the function. Easy, isn't it?

More stuff you need to know about packages

There is a bit more you need to know how packages work:

Packages can be nested

You can have packages that live inside of other packages. Like here:



Calling function in sub-packages is quite straight forward:

```
mother.anna.my_fun();
mother.max.my_fun();
```

You always need to call functions in packages with all the packages

Ok, sounds weird at first. Here is the thing. Even if you call a function that lives in the same package as the function calling it, you need to write all the packages.

For example: In the code, you will find two more function in the `mother.anna` package: `add_numbers` and `my_other_fun`. `my_other_fun` wants to call `add_numbers`.

This would not work:

```
function my_other_fun()
result = add_numbers(1, 3);

fprintf('One and three is: %d\n', result);

end
```

But this does:

```
function my_other_fun()
result = mother.anna.add_numbers(1, 3);

fprintf('One and three is: %d\n', result);

end
```

Finally

Play around with the package system. Write your own small toolbox and try to get the grip of it!

About Classes and Object-Oriented Programming Part 1

So, you made it to the scary topic. But let me assure you: the basic concepts of object-oriented programming are really easy. And you already know most of the really important stuff - You just have not realized it yet.

The main problem of understanding object-oriented programming is that it is so extremely powerful that you can do very very complex stuff with it. And for developers, this super complex stuff is so awesome that they will talk about that.

We don't do this here. We will keep it very very simple.

Let's create our first class

First, create a new folder that will contain the packages, classes, functions etc. of this tutorial.

Navigate into the folder you just created and create another folder for the "toolbox" you are about to write. For this tutorial, I will assume that this new folder is called `tut_toolbox`.

So, after you have created the folder for your new toolbox, create a new package folder called `+example01`.

If you do a right-click on the `+example01` folder and then select *New File*, it offers to create a class for you. If you do this and call your new class `Human.m`, it will create this for you:

```
classdef Human
 %HUMAN Summary of this class goes here
 %   Detailed explanation goes here

 properties
 end

 methods
 end

end
```

So, what do we have here:

1. The first line basically says: "I want a class that is called *Human*".
    1. As a convention, all classes begin with a capital letter, while all function names start with a lower case letter!
2. We have an `end` statement at the end. You already know this from your functions and `for` loops. Everything between the `classdef` and the last `end` statement belongs to the description of the class.
3. Within the `classdef` description, we have empty blocks for something called `properties` and something else called `methods`.

Congratulations, you have just created your first class. Yes, it does not do anything at the moment, but we will change that soon.

Adding methods

Up to now, our class does not do anything. But you have noticed the `methods` block in the file that Matlab created for you. This is, where the class's methods go.

Methods are functions of a class

So, lets add a simple method to the class:

```
classdef Human
 properties
 end

 methods
   function hello(obj)
     disp('Hi!');
   end %function
```

(continues on next page)

(continued from previous page)

```
  end

end
```

If you compare it to the code we started with, you see that we added a normal Matlab function definition inside the `methods` block. You can also see that the function takes one argument called `obj`. Just ignore this for the moment and just treat the function as if it had no argument at all.

So, it should be obvious for you what this function should do. So let's try that:

```
%% clear and restore path...
clear all global

restoredefaultpath

%% add the toolbox to the path...
addpath('tut_toolbox');

%% call the class function...
example01.Human.hello();
```

If we do this, Matlab is going to complain:

```
The class example01.Human has no Constant property or Static method named 'hello'.
```

About Classes and Instances

So, why did we get an error message? In order to understand this, you need to know the difference between *classes* and *instances*:

What is a Class?

The easiest way to imagine what a class is, is to imagine it as a plan or template.

Let's say, we want to build a car. In this case, the *class* would be the plans and all instructions on **how to build a car**. It is **not** the actual car itself.

But just as you can use the plans and instructions to build a concrete car (or two, three, thousands), we can use a *class* to create a concrete *instance* (or two, three, thousands).

What is an Instance?

An *instance* is the concrete object built by using the definitions and descriptions in a *class*. In our car analogy, it would be the actual car.

If you think about it, this distinction is really important. The actual car can do stuff like accelerated, brake, steer to the left or right. The plan of the car cannot do that, but it describes how this is done.

This is the reason why we got this error in the last section. We basically asked the plan to do something. But the plan cannot do something. It just knows how to do it. We need to build an actual object (or instance) with the help of that plan. The resulting thing (instance, car, whatever) can then do the stuff that is defined in the plan.

How to get an instance

Here is how you get an instance of a class:

```
my_first_human = example01.Human();
```

You see, its really easy: You just write down the name of the class, including all the packages, of course, and the put brackets and assign the result to a variable. Now `my_first_human` points to an instance of `Human`.

Now, we can call the *methods* of the instance like this:

```
my_first_human.hello()
```

So, if you want to call a method of an instance, you first type the name of the instance (**not** the class!), the a dot (`.`) and then the name of the method. Easy, right?

About properties

Up to now, our new class can do stuff (it can say "hi"). But it would also be good if we could also attach some data to an instance of a class. This is what properties are for.

Properties are variables of a class

Or, if you would like to compare it to the structures that you know from FieldTrip:

Properties are the fields of a class

So, let's add a property to our class:

```
classdef Human
  properties
    name
  end

  methods
    function hello(obj)
      disp('Hi!');
    end %function
  end

end
```

Now, create an instance of this class:

```
my_first_human = example01.Human();
```

and look what is inside:

```
>> my_first_human

my_first_human =

  Human with properties:

    name: []
```

Matlab tells us that our Human now has a property called name. We can use it now as we would use fields of structures:

```
my_first_human.name = 'Adam';
my_first_human.name
```

Properties are specific to instances, not classes

It is really important to understand this. You can create as many instances of a class as you like. But assigning a property of calling a method only affects the instance and not the class!

Here is an example:

```
my_first_human = example01.Human();
my_first_human.name = 'Adam';

my_second_human = example01.Human();
my_second_human.name = 'Eve';

my_first_human
my_second_human
```

You see that both variables point to an instance of the class Human, so they look alike. But their data is different.

Let's do something with the property

One of the incredibly nice and powerful things about Object-Oriented Programming is the fact that the methods of a class can operate on its properties.

For example: Our Human class now has a property called name. Let's use it to write a function that prints out the name:

```
classdef Human
  properties
    name
  end

  methods
    function hello(obj)
      disp('Hi!');
    end %function

    function say_name(obj)
      disp(['My name is ' obj.name '!']);
    end %function
  end

end
```

You see, we created another function called say_name. The function does only one thing:

```
disp(['My name is ' obj.name '!']);
```

Do you remember that I told you to forget about the obj parameter earlier? Now we use it.

A method of a class can have any number of parameters. **But** the first parameter always receives the current instance of the class. By convention, this first parameter is always called `obj`. The method can then use this parameter to access the instance's properties and call its functions. Just like we do here to access the **instance's** `name` property.

Try it out:

```
my_first_human.say_name();
my_second_human.say_name();
```

About Classes and Object-Oriented Programming Part 2

Welcome to part 2 about classes and objects.

You have already written your first class. You know what is meant when we talk about an instance of a class. You know how to add properties and methods to a class.

Great, let's go on.

Preparations

We are going to do some changes to the `Human` class that we developed during the first part of the Classes-And-OOP tutorial. These changes make it impossible to use the class with the example scripts that we have written so far. So, i would suggest you copy the class and give it a new name. You can call it as you wish, but I will assume that it is called `AdvancedHuman`.

At the beginning, this class is a mere copy of the `Human` class. So it look like this:

```
classdef AdvancedHuman
  properties
    name
  end

  methods
    function hello(obj)
      disp('Hi!');
    end %function

    function say_name(obj)
      disp(['My name is ' obj.name '!']);
    end %function
  end

end
```

Naming conventions

Especially in collaborative projects, it is important to agree on some convention on how to name things. In o_ptb as well as in this tutorial, the following convention is used:

1. Variables, Properties, Functions and Methods are always written in lower case. Words are separated by an underscore:
    1. `function hello(obj)`
    2. `function say_name(obj)`
    3. `name = 'Adam';`
    4. `postal_code = '12345';`
2. Classes are always written with a capital first letter followed by lower case letters. If a class name consists of two or more words, a capital letter marks the beginning of a new word:
    1. `classedef AdvancedHuman`

Constructors

If we want to create an instance of an `AdvancedHuman` and give it the name "Max", it is a two-step process:

```
my_human = examples01.AdvancedHuman();
my_human.name = 'Max';
```

But this does not really make sense, does it? It does not make sense, because a human always has a name. So, it would be much better to supply the name when we create the instance.

This is where **Constructors** come in. Every class has a constructor. It is a special method that gets called when you create a new instance of a class.

You might be a bit confused now because you do not see such a method in the code. You are right. Matlab is so nice to just add an empty constructor when we do not provide one.

Adding a constructor is really easy. It is just an extra method like this:

```
function obj = ClassName()
```

In order to be a valid constructor, a method needs to follow three requirements:

1. It must have the **exact same name** as the class.
2. It **must** return a variable called `obj` and nothing else.
3. Contrary to all other methods, it does not take `obj` as its first parameter.

So, let's add a simple constructor to our class:

```
classdef AdvancedHuman
  properties
    name
  end

  methods
    function obj = AdvancedHuman()
      disp('Creating a nice Human for you.');
```

(continues on next page)

(continued from previous page)

```
    end %function

    function hello(obj)
      disp('Hi!');
    end %function

    function say_name(obj)
      disp(['My name is ' obj.name '!']);
    end %function
  end

end
```

You see that the constructor does not really do something now other than displaying a message. But lets try it:

```
>> my_human = example01.AdvancedHuman();
Creating a nice Human for you.
```

Ok, that worked well. Let's implement the functionality I described above: We want the constructor to take the name and automatically put it into the correct property of the instance.

Here is the code:

```
classdef AdvancedHuman
  properties
    name
  end

  methods
    function obj = AdvancedHuman(name)
      fprintf('Creating a nice Human called %s for you.\n', name);
      obj.name = name;
    end %function

    function hello(obj)
      disp('Hi!');
    end %function

    function say_name(obj)
      disp(['My name is ' obj.name '!']);
    end %function
  end

end
```

This is what happens now:

1. The constructor now requires a parameter which is called name.
2. It uses the value of that parameter in the next line to display the message.
3. It the assigns the value to the property called `name` of the instance, which it can access via the `obj` variable.

I deliberately chose to use the same name for the parameter as for the property to demonstrate how both of them are accessed:

1. `name` is the parameter of the method.
2. `obj.name` is the property called "name" of the current instance.

Now you can do this:

```
my_human = example01.AdvancedHuman('Adam');
my_human.say_name();
```

Access rights

At the moment, we can still do this:

```
my_human = example01.AdvancedHuman('Adam');
my_human.name = 'Eve';
```

I don't think this makes much sense because a name is given one time at birth and does not change (yes, there are exceptions to this.).

Luckily, you can set access rights for properties and methods which defines who can call the methods or write/read to the property.

Matlab knows three levels of access rights:

1. Public: The method or property can be accessed from anywhere.
2. Private: The method or property can only be accessed from methods of that specific class.
3. Protected: Like private but access is also allowed from sub-classes (more on that later).

For properties, you can set these rights separately for reading and writing to the property.

So, in our case, our code now looks like this:

```
classdef AdvancedHuman
  properties (SetAccess=private)
    name
  end

  methods
    function obj = AdvancedHuman(name)
      fprintf('Creating a nice Human called %s for you.\n', name);
      obj.name = name;
    end %function

    function hello(obj)
      disp('Hi!');
    end %function

    function say_name(obj)
      disp(['My name is ' obj.name '!']);
    end %function
  end

end
```

We have restricted access to the property `name` so that it can only be written to from methods of the class. So, assigning a value as in the constructor still works because the constructor is a member of the class.

But this does not work anymore:

```
my_human = example01.AdvancedHuman('Adam');
my_human.name = 'Eve';
```

About Classes and Object-Oriented Programming Part 3

Welcome to part 3 about classes and objects.

You now know about constructors and access rights.

Let's tackle the last big challenge: Inheritance

Inheritance is a very powerful tool in Object-Oriented Programming. Because it is possible to do very complex things with it, it might seem very complex. I promise to make it as simple as possible.

So, let's start with an example....

What we want to do

So, we already have a class describing our `AdvancedHuman`. It has a name and can do two things: 1. say hello and 2. say its name.

However, you might agree that this is a very generic class. There might be special groups of humans who would do things differently. But they would still be humans, of course...

Let's make a pirate

For instance, pirates are a special kind of humans, because instead of saying 'Hi', they say 'ARRRRGGGHHH'.

Create a new class in the same package as the `AdvancedHuman` called Pirate.m

Here is the minimum code to start:

```
classdef Pirate < example01.AdvancedHuman

  methods
    function obj = Pirate(name)
      obj = obj@example01.AdvancedHuman(name);
    end %function
  end %methods

end
```

Ok, let's go through it line by line:

1. `classdef Pirate < example01.AdvancedHuman`: You already know the first part. We define a new class. The second part (`< example01.AdvancedHuman`) means that we do not create a class from scratch but instead we take the `AdvancedHuman` class and start from there. This means that our new class already has **all the methods and all the properties** defined in the `AdvancedHuman` class!
2. `function obj = Pirate(name)`: We also need a constructor for our new class because we want our constructor to take a parameter.
3. Remember that the constructor initializes all the properties. In our case, we would need to assign the value of the `name` parameter to the `obj.name` property of the classe. We could just write something like this:

```
function obj = Pirate(name)
    obj.name = name;
end %function
```

But this has some disadvantages:

1. We repeat code that is already written. This is very bad in general!
2. Maybe `AdvancedHuman` does some more initialization work? We don't know. Or maybe `AdvancedHuman` receives an update at some point in the future that requires it to do some extra work in its constructor.

   So, instead of doing the work of the class we inherit from ourselves, we let it do the work.

4. `obj = obj@example01.AdvancedHuman(name);`: This is how you call the constructor of the "superclass". The "superclass" is the class your new class inherits from.

Now we can write something like this:

```
my_pirate = example01.Pirate('Adam');
my_pirate.say_name();
my_pirate.hello();
my_pirate.name
```

We now have a class called `Pirate` that has all the methods and properties of an `AdvancedHuman` without repeating the code.

But this is quite boring. We wanted our pirate to do something different than the ordinary `AdvancedHuman` when we ask him to say hello. This is how it works:

```
classdef Pirate < example01.AdvancedHuman

  methods
    function obj = Pirate(name)
      obj = obj@example01.AdvancedHuman(name);
    end %function

    function hello(obj)
      disp('AAARRRRGGGHHHH');
    end %function
  end %methods

end
```

Now just execute the same code again and see the difference!

You will see that instead of using the `hello` method of the `AdvancedHuman` class, the `Pirate` class now uses the `hello` method we defined here.

Why this is useful

You might ask yourself: "What is the reason for doing all this?". It is quite simple:

We can define one base class (in this case `AdvancedHuman`) that has all the properties and methods that we need for a specific purpose. We can then create a function like this that works with instances of that class:

```
function group_hello(humans)
for i = 1:length(humans)
  cur_human = humans{i};
  cur_human.say_name();
  cur_human.hello();
end %for
end
```

You can see that this function does not care, what kind of `AdvancedHuman` it gets. It might be a direct instance of `AdvancedHuman` or it might be an instance of a class that inherited from `AdvancedHuman`.

Take a look at this script that calls the `group_hello` function:

```
%% clear and restore path...
clear all global

restoredefaultpath

%% add the toolbox to the path...
addpath('my_toolbox');

%% get some pirates and humans
my_pirate = example01.Pirate('Adam');

my_first_human = example01.AdvancedHuman('Mary');
my_second_human = example01.AdvancedHuman('Eve');

%% send them all to the group_hello function
all_humans = {my_pirate, my_first_human, my_second_human};

example01.group_hello(all_humans);
```

This script create one `Pirate` and two `AdvancedHuman`s and sends these to the `group_hello` function which then executes both methods of each instance provides in the cell-array.

The important concept behind this is that we have created a hierarchical order between the two classes: Every `Pirate` is also an `AdvancedHuman`. So a function can require to be provided with an instance (or instances) of `AdvancedHuman` but it would also work with a `Pirate`.

About "Handle Classes"

All classes used in p_ptb are so-called "handle classes". What does this mean?

When you create an object of a handle class and assign it to a variable like this:

```
my_object = MyHandleClass();
```

What gets stored in the variable `my_object` is **not the object itself** but a "handle" to it. A "handle" is just an address within the computer's memory. So the variable `my_object` stores **where the object is**. It **does not store the object itself!**

What are the implications of this? Let's create a simple handle class and put it in the package `+handle_class_example`:

```
classdef MyHandleClass < handle
  properties
    number
  end
end
```

So, we have a simple class with only one property.

Let's create a second class but without the handle:

```
classdef MyNormalClass
  properties
    number
  end
end
```

The only difference between the two is that `MyHandleClass` inherits from `handle` while `MyNormalClass` does not.

If you run this, the output will be as you would expect:

```
>> %% get a normal class
normal_class1 = handle_class_example.MyNormalClass();
normal_class1.number = 1;

%% copy it by assignment
copy_of_normal_class1 = normal_class1;
copy_of_normal_class1.number = 100;

%% see what it is both classes
normal_class1
copy_of_normal_class1

normal_class1 =

  MyNormalClass with properties:

    number: 1.0000e+000


copy_of_normal_class1 =

```

(continues on next page)

(continued from previous page)

```
  MyNormalClass with properties:

    number: 100.0000e+000
```

If we do the same thing again, but this time use the `MyHandleClass`, this happens:

```
>> %% get a handle class
handle_class1 = handle_class_example.MyHandleClass();
handle_class1.number = 1;

%% copy it by assignment
copy_of_handle_class1 = handle_class1;
copy_of_handle_class1.number = 100;

%% see what it is both classes
handle_class1
copy_of_handle_class1

handle_class1 =

  MyHandleClass with properties:

    number: 100.0000e+000


copy_of_handle_class1 =

  MyHandleClass with properties:

    number: 100.0000e+000
```

So, whats different here? By assigning a new value to the copy of the object, we also changed what we get when we refer to the original one. This is, as I wrote above, because `handle_class1` does not store the object itself but only its handle, i.e. its address in memory. `copy_of_handle_class1 = handle_class1;` assigns the address of the object `handle_class1` is pointing at to the variable `copy_of_handle_class1`. As it is the address that is exchanged, both variables point at the **same** object!!

So, how can you do a real copy of a handle class? First, you need to modify the class a little bit:

```
classdef MyHandleClass < handle & matlab.mixin.Copyable
  properties
    number
  end
end
```

And now you can do this:

```
>> %% get a handle class
handle_class1 = handle_class_example.MyHandleClass();
handle_class1.number = 1;

%% copy it by copy
copy_of_handle_class1 = copy(handle_class1);
```

(continues on next page)

(continued from previous page)

```
copy_of_handle_class1.number = 100;

%% see what it is both classes
handle_class1
copy_of_handle_class1

handle_class1 =

  MyHandleClass with properties:

    number: 1.0000e+000


copy_of_handle_class1 =

  MyHandleClass with properties:

    number: 100.0000e+000
```

So, the bottom line is: Always use the `copy` function when you want to really copy a handle class!

ALL classes of o_ptb are handle classes. ALL classes can be copied

2.2.2 About the o_ptb

About o_ptb

So, let's use all your new knowledge about packages and object-oriented programming to understand o_ptb.

What is o_ptb?

o_ptb is a so-called "class library". A "class library" is nothing else than a toolbox (or function library) that makes use of classes to provide its functionality.

The main purpose of o_ptb is to make using the Psychophysics Toolbox and the Vpixx/Datapixx system easier and more coherent.

Both toolboxes (i.e. Psychtoolbox and Datapixx) are very powerful but are very low level. This property makes the hard to use correctly if you do not know exactly how especially the Datapixx system works. And it makes your code look really ugly and not intuitive. In addition, you might have noticed that you would repeat lots of code when you develop your experiment using Psychtoolbox and Datapixx. Repeating code, especially low-level code is very undesirable because it makes it easier to do mistakes and it makes it harder to correct them.

o_ptb provides functionality to overcome this by putting a thin layer of abstraction on top of both toolboxes.

What does "abstraction" mean? Glad you asked: Abstraction means that you hide low-level code and provide functions that do common jobs with an easier interface. Take a car for example: If you want to construct or repair a car, you need to know a lot about how it works on the low-level. Like how the engine works, the gears, the brakes etc. Accelerating, for instance, is quite a complex thing to do for a car: It needs to inject just the right amount of fuel, it needs to monitor the ignition and all kind of other stuff. Luckily, if you just want to drive a car, this complexity is hidden from you. You just press down the accelerator and your car goes faster.

The design principles of o_ptb

When I planned how to develop o_ptb, I used the following principles:

1. Be as intuitive as possible. If a function say "draw", it draws.
2. Allow all Psychtoolbox functions that do something visual to still be used.
3. User must not use Datapixx functions directly.
4. Code written by users must run with and without a Datapixx and should behave the same.

Getting Started with o_ptb

Initializing o_ptb

The initialization process of o_ptb includes the following steps:

1. Make sure your workspace and your path are clean.
2. Add the top-level folder of o_ptb to your path.
3. Use o_ptb to initialize PTB.
4. Configure o_ptb.
5. Create a *+o_ptb.PTB* object which is the "manager object".
6. Initialize the subsystems (visual, audio, trigger, response) you need.

Make sure your workspace and your path are clean

It is **really** important to start out with a clean Matlab path and workspace. A lot of problems can happen if you have multiple versions of Psychtoolbox in your path, just to give an example. So, I advise you to do this at the beginning of every script:

```
%% clear
clear all global
restoredefaultpath
```

Add the top-level folder of o_ptb to your path

This should be an easy one for you:

```
%% add the path to o_ptb
addpath('/home/th/git/o_ptb/') % change this to where o_ptb is on your system
```

This adds the top-level folder of o_ptb to your path. The only thing that is in there is actually a package called `o_ptb`, which houses all the functions and classes.

Use o_ptb to initialize Psychtoolbox

Unfortunately, Psychtoolbox does not provide a nice initialization function for its paths like FieldTrip or obob_ownft. Fortunately, o_ptb provides such a function for you:

```
%% initialize the PTB
o_ptb.init_ptb('/home/th/git_other/Psychtoolbox-3/'); % change this to where PTB is on␣
↪your system
```

You see that we are calling the function called *+o_ptb.init_ptb()* in the package `o_ptb`. You can also call that function without any argument in which case it will search your current path and subfolders for Psychtoolbox. Another possibility is to set *O_PTB_PTB_FOLDER* Environment Variable.

Configure o_ptb

Next, we need to configure how o_ptb should behave. In order to do that, we need to get a PTB_Config object:

```
ptb_cfg = o_ptb.PTB_Config();
```

+o_ptb.PTB_Config is a class, so this line creates an object of that class. If you just type `ptb_cfg` now on the command line, this is, what you will see:

```
>> ptb_cfg

ptb_cfg =

  PTB_Config with properties:

                fullscreen: 1
              window_scale: 1.0000e+000
          draw_borders_sbg: 1
           flip_horizontal: 0
                hide_mouse: 1
          background_color: 127.0000e+000
            skip_sync_test: 0
            force_datapixx: 0
             crappy_screen: 0
     psychportaudio_config: [1×1 o_ptb.PTB_subconfigs.PsychPortAudio]
   keyboardresponse_config: [1×1 o_ptb.PTB_subconfigs.KeyboardResponse]
      datapixxaudio_config: [1×1 o_ptb.PTB_subconfigs.DatapixxAudio]
   datapixxresponse_config: [1×1 o_ptb.PTB_subconfigs.DatapixxResponse]
                  defaults: [1×1 o_ptb.PTB_subconfigs.Defaults]
           internal_config: [1×1 o_ptb.PTB_subconfigs.PTBInternal]
```

You see that `o_ptb` has a bunch of properties you can configure: You also see that every property already has a default value. If you want to know what all these properties mean, you can just type:

```
help ptb_cfg
```

on the command line. And yes, this would be equivalent to typing `help o_ptb.PTB_Config`.

The `ptb_cfg` object is very similar to a *cfg* structure you know from FieldTrip with one important exception: You cannot add new fields/properties.

You can also see that the `ptb_cfg` object has some properties that are objects themselves, for example the `psychportaudio_config` property. These properties configure the subsystems. You can leave them alone for now. We will look at them later.

For now, we want our system to:

1. Not show the experiment full screen
2. Thus scale the window down to 1/5 of its original size (which is alway 1920*1080).
3. Not complain if it fails a sync test.
4. Not hide the mouse.

To achieve this, we do this:

```
ptb_cfg.fullscreen = false;
ptb_cfg.window_scale = 0.2;
ptb_cfg.skip_sync_test = true;
ptb_cfg.hide_mouse = false;
```

Create a o_ptb.PTB object which is the "manager object"

The central "hub" for all things related to o_ptb is an object of the *+o_ptb.PTB* class. It handles the screen, the audio, trigger and response subsystem. Long story short: It does all the dirty work behind the scenes.

+o_ptb.PTB is a special class in the sense that there must always be **one and only one** object of it. There can be many variables with different names pointing to it. In order to achieve this, I needed to use a small trick.

In order to get the `o_ptb.PTB` object, you need to do this:

```
ptb = o_ptb.PTB.get_instance(ptb_cfg);
```

You can see that this is quite different from how we normally get an instance (i.e., an object) of a class. Normally you would call it like: `o_ptb.PTB(ptb_cfg);`. The reason for doing it with the *+o_ptb.PTB.get_instance()* method is that this method takes care of the requirement that there can only be one *+o_ptb.PTB* around.

If you take a look at the *reference for +o_ptb.PTB* you see that your new object provides some specific and handy methods to do all kinds of things you need to do when developing an experiment. It can take care of setting up the screen for you according to the configuration you provided. If it finds a Datapixx, it will use that to improve timing. It can also initialize the audio, trigger and response system for you. If it finds a Datapixx, it will use it. Otherwise, it will use your keyboard for responses, your sound card for audio and (for the moment) just write the trigger values to the command window.

You can also see that it provides you with some properties that are important when you want to draw stuff on the screen. You need the window handle (also called window_pointer in Psychtoolbox)? Take it from here: `+o_ptb.PTB.win_handle`. You need to know the height and width of your screen in pixels? It can do that, too!

Initialize the subsystems (visual, audio, trigger, response) you need

The final step of initialization is to setup the subsystems that you need. At the moment, we only want to draw stuff on the screen, so here we go:

```
ptb.setup_screen;
```

This step will take some time. You should see a small PTB window pop up at the top-left of your screen.

When the method returns, we should take a look at the properties of `ptb`:

```
>> ptb

ptb =

  PTB with properties:

              win_handle: 10.0000e+000
                win_rect: [0.0000e+000 0.0000e+000 1.9200e+003 1.0800e+003]
           flip_interval: 16.6636e-003
             width_pixel: 1.9200e+003
            height_pixel: 1.0800e+003
    using_datapixx_video: 0
            scale_factor: 200.0000e-003
```

Do you notice that `height_pixel` is 1080 and `width_pixel` is 1920? But hey, the window that has just opened is **way** smaller!!

Yes, you are right. This is another neat trick of Psychtoolbox: It scales down the original fullscreen window to the small window you have just created. This means that when you want to draw something on the window, you do this in the original resolution and PTB handles all the scaling for you:

ATTENTION

Only use this feature when you are developing! If you are running your experiment on a computer that has a different resolution (or you want your screen to be at a lower one), you need to tell o_ptb that resolution in the configuration step:

```
ptb_cfg.internal_config.final_resolution = [1024 768];
```

How to draw stuff on the screen

So, you have mastered the initialization and configuration of o_ptb. Now it's time to draw things on the screen.

Drawing using native Screen commands

As I have stated before, you can use all the Psychtoolbox commands to draw stuff on the screen. The `Screen()` function needs the window_handle which you can get from the *+o_ptb.PTB* instance:

```
Screen('FillRect', ptb.win_handle, [0 0 0], [200 200 300 300]);
```

The next thing you want to do to see you rectangle on the screen is a `Screen('Flip')`.

Screen('Flip') is the only command you MUST NOT USE!

In order to get all the timing and added functionality right, you must do:

```
ptb.flip();
```

ptb.flip() takes the same arguments as the `Screen('Flip')` function and also returns the same values.

So, if we want our rectangle to appear on the screen after one second and save the timestamp in a variable, we would need to do this:

```
Screen('FillRect', ptb.win_handle, [0 0 0], [200 200 300 300]);
timestamp = ptb.flip(GetSecs()+1);
```

Using the o_ptb convenience function for `Screen`

o_ptb provides a convenience function for you so you do not have to always provide the window handle:

```
ptb.screen('FillRect', o_ptb.constants.PTB_Colors.white, [300 300 400 400]);
timestamp = ptb.flip(GetSecs()+1);
```

You can use **all** drawing related `Screen` functions like this. The parameters are the same, except for the omission of the window_handle parameter. The return values are identical as well.

Drawing a o_ptb provided stimulus

As another convenience, o_ptb also provides certain often-used visual stimuli that you can draw on the screen. These are provided as classes.

You can find the stimuli in the package `+o_ptb.stimuli.visual`. The easiest one is the fixation cross:

```
fix_cross = o_ptb.stimuli.visual.FixationCross();
ptb.draw(fix_cross);
ptb.flip();
```

So, you create an object of the stimulus class you want to draw. You call *ptb.draw* to draw it and the you flip it.

Please note that you only need to create the stimulus object once. It can be used as often as you like!

Here is how you draw text:

```
hello_world = o_ptb.stimuli.visual.Text('Hello World!');
ptb.draw(hello_world);
ptb.flip();
```

The `hello_world` is an object that has some interesting properties:

```
>> hello_world

hello_world =

  Text with properties:

               size: 46.0000e+000
              style: 0.0000e+000
               font: 'Arial'
```

(continues on next page)

(continued from previous page)

```
                sx: 'center'
                sy: 'center'
            wrapat: 80.0000e+000
             color: 0.0000e+000
          vspacing: 1.0000e+000
              text: 'Hello World!'
    destination_rect: [0.0000e+000 0.0000e+000 1.9200e+003 1.0800e+003]
```

You can change those properties if you like. Let's say, you want to increase the size of the font:

```
hello_world.size = 90;
ptb.draw(hello_world);
ptb.flip();
```

There are many more stimuli provided by o_ptb:

```
gabor = o_ptb.stimuli.visual.Gabor(400);
gabor.frequency = .01;
gabor.sc = 60;
gabor.contrast = 120;

ptb.draw(gabor);
ptb.flip();
```

You can also draw an image from an image file:

```
smiley = o_ptb.stimuli.visual.Image('smiley.png');

ptb.draw(smiley);
ptb.flip();
```

Ok, that one is a little big. So let's scale it:

```
smiley.scale(0.5);

ptb.draw(smiley);
ptb.flip();
```

We can also move the image:

```
smiley.move(150, 200);

ptb.draw(smiley);
ptb.flip();
```

Using Triggers and Sound

Triggers and Sound are handled entirely different from how you are used to. The reason for this is that depending on whether you want to use the Datapixx system or not, the commands you normally have to use are completely different. o_ptb changes this by providing a unified interface for you.

In fact, you do not need to worry whether a Datapixx is connected or not. o_ptb tries to find it at the beginning and if it is connected, it will use it. If not, it will automatically fall back to using PsychPortAudio for the sound and just print the triggers in the command window (for now).

This is the strength of abstraction: The user does not need to care whether a Datapixx is connected or not. The code will run in either case. And if you want to support other sound or trigger systems in the future, all you need to do is write a new class doing the low-level work and plug it into o_ptb.

Introduction to how Triggers and Sound works

You might ask yourself, why Triggers and Sounds are both handled in this section. The answer is simple: o_ptb follows the way it is implemented in the Datapixx system. And for the Datapixx system, triggers and sounds are basically the same.

The Datapixx provides a buffer (i.e. some internal memory) to upload your sound data to. After you have done that, you can tell it to start playing that sound either at once or at the next flip. Triggers are basically handled like an additional channel to the sound. So you can also upload your trigger data and "play" it at once or at the next flip.

Initializing

The code to initialize everything is really similar to what you already know:

```
%% clear
clear all global
restoredefaultpath

%% add the path to o_ptb
addpath('/home/th/git/o_ptb/') % change this to where o_ptb is on your system

%% initialize the PTB
o_ptb.init_ptb('/home/th/git_other/Psychtoolbox-3/'); % change this to where PTB is on␣
↪your system

%% get a configuration object
ptb_cfg = o_ptb.PTB_Config();

%% do the configuration
ptb_cfg.fullscreen = false;
ptb_cfg.window_scale = 0.2;
ptb_cfg.skip_sync_test = true;
ptb_cfg.hide_mouse = false;

%% get o_ptb.PTB object
ptb = o_ptb.PTB.get_instance(ptb_cfg);

%% init audio and triggers
ptb.setup_screen;
```

(continues on next page)

(continued from previous page)

```
ptb.setup_audio;
ptb.setup_trigger;
```

You can see that the only real difference is these two lines:

```
ptb.setup_audio;
ptb.setup_trigger;
```

Now we have our visual, audio and trigger system ready to use. Please note that it is not necessary to setup the visual system if you only want to present sounds and/or triggers. But we are going to need it later, so I leave it in.

Presenting a wav file

In order to send sounds to the audio system, we need to get an object representing that sound. At the moment, o_ptb offers two kinds of audio stimuli: You can get it from a wav file or supply it via a matrix.

Let's keep it simple at first and load the wav file that you find in the folder:

```
my_sound = o_ptb.stimuli.auditory.Wav('da_40.wav');
```

Now we have the auditory stimulus object. In order to present it, we need to do a three step process:

1. Call *ptb.prepare_audio* to tell o_ptb that you want to use it. You can call this function many times if you want to present multiple stimuli in a row.
2. Once you have prepared all you sound stimuli, you need to call *ptb.schedule_audio*. This uploads all the sound data to the Datapixx or sound card and prepares everything to play it.
3. Tell o_ptb when to play it. Either use *ptb.play_on_flip* to automatically play it the next time you call *ptb.flip*. Or you can use *ptb.play_without_flip* to play it at once.

Here is the code:

```
%% prepare sound
ptb.prepare_audio(my_sound);

%% schedule sound
ptb.schedule_audio;

%% play at once
ptb.play_without_flip;
```

Presenting with a delay and two sounds in a row

ptb.prepare_audio takes two additional parameters. The first is the delay in seconds after which to present the sound. So, if you want the sound to start 500ms after you called *ptb.play_without_flip* or after the flip, you do this:

```
ptb.prepare_audio(my_sound, 0.5);
ptb.schedule_audio;
ptb.play_without_flip;
```

The third parameter allows you to prepare a second sound while holding the first. So, if you want to play the sound twice, once without delay and once after 600ms, you do this:

```
ptb.prepare_audio(my_sound);
ptb.prepare_audio(my_sound, 0.6, true);

ptb.schedule_audio;
ptb.play_without_flip;
```

Create your own sounds

If you do not want to load your sounds from a wav file but rather supply the data directly, you can use the +o_ptb.stimuli.auditory.FromMatrix stimulus:

```
%% create a sine wave and make a sound object
s_rate = 44100;
freq = 440;
amplitude = 0.1;
duration = 1;

sound_data = amplitude * sin(2*pi*(1:(s_rate*duration))/s_rate*freq);

sin_sound = o_ptb.stimuli.auditory.FromMatrix(sound_data, s_rate);

%% play it
ptb.prepare_audio(sin_sound);
ptb.schedule_audio;
ptb.play_without_flip;
```

Adding triggers

Adding triggers is really easy and very similar to using sounds:

```
ptb.prepare_audio(my_sound);
ptb.prepare_trigger(1);

ptb.schedule_audio;
ptb.schedule_trigger;

ptb.play_without_flip;
```

ptb.prepare_trigger takes the same arguments as *ptb.prepare_audio*. So you can do this:

```
ptb.prepare_audio(my_sound);
ptb.prepare_trigger(1);

ptb.prepare_audio(my_sound, 0.5, true);
ptb.prepare_trigger(2, 0.5, true);

ptb.schedule_audio;
ptb.schedule_trigger;

ptb.play_without_flip;
```

Playing sound and triggers when a visual stimulus appears

Of course, you would like to synchronize the onset of your sounds and triggers to when some visual stimulus appears on the screen. This is done by using *ptb.play_on_flip* instead of *ptb.play_without_flip*. *ptb.play_on_flip* does not do anything until you issue *ptb.flip*:

```
hello_world = o_ptb.stimuli.visual.Text('Hello World!');
ptb.draw(hello_world);

ptb.prepare_audio(my_sound);
ptb.prepare_trigger(1);

ptb.prepare_audio(my_sound, 0.5, true);
ptb.prepare_trigger(2, 0.5, true);

ptb.schedule_audio;
ptb.schedule_trigger;

ptb.play_on_flip;

ptb.flip(GetSecs + 1);
```

Things to keep in mind to keep the timing right

Some of the functions presented here take some time to run while others are very quick. For example, creating a stimulus object can take quite a long time. So it is a good idea to create all your stimulus objects after the initialization and **not** when running your trial.

It is also a good idea to call all the prepare and schedule functions in sections that are not time critical.

How to get responses

Unifying the way how to get responses from the Datapixx (when present) and the keyboard (when the Datapixx is not present) is a little tricky because the response pad of the Datapixx has 4 buttons with different colors while your keyboard has keys.

To get around this issue, o_ptb does something called "button mapping". This means that you define a response, give it a name and assign a Datapixx button and a keyboard key to it.

This is done during the configuration process:

```
%% get a configuration object
ptb_cfg = o_ptb.PTB_Config();

%% configure button mappings
ptb_cfg.datapixxresponse_config.button_mapping('target') = ptb_cfg.datapixxresponse_
↪config.Green;
ptb_cfg.keyboardresponse_config.button_mapping('target') = KbName('space');

ptb_cfg.datapixxresponse_config.button_mapping('other_target') = ptb_cfg.
↪datapixxresponse_config.Red;
ptb_cfg.keyboardresponse_config.button_mapping('other_target') = KbName('RightShift');
```

We define two responses here called “target” and “other_target”. We the assign keys and buttons to it.

You can assign the same button or key to multiple targets!

You can then use the *ptb.wait_for_keys* function to wait until one or both responses are issued. For example:

```
ptb.wait_for_keys('target', GetSecs+1)
```

This line waits up to one second for a press of the “target” response. So, if you have a Datapixx, it will wait for the green button to be pressed. If not, it waits for the space key.

The function returns instantly as soon as the response is pressed. It returns a cell of the responses issued during the call. If no response was given, it returns after the timeout and returns an empty cell.

You can also wait for multiple responses at the same time:

```
ptb.wait_for_keys({'target', 'other_target'}, GetSecs+10)
```

Showing Movies

A movie is basically a set of images that need to be displayed at very specific and regular intervals. In `o_ptb`, a movie is nothing else than a `+o_ptb.stimuli.visual` stimulus with some extra convenient methods.

Note: If you want to display a silent movie simply as a distraction to the participants (i.e., in a passive listening task), it is much easier to just play the movie with VLC or any other movie player while the auditory part of the experiment is running.

Note: `o_ptb` only supports playing the video part of the movies. You cannot use the audio streams at the moment.

Loading the movie

In order to prepare the movie, instantiate an object of *+o_ptb.+stimuli.+visual.Movie*, providing the filename of the movie:

```
my_movie = o_ptb.stimuli.visual.Movie('movie.avi');
```

Displaying the movie

We can now start the movie internally:

```
my_movie.start();
```

Now the movie is ready to be displayed. We now need to these three things in a loop:

1. Request a new frame of the movie and check whether one is available.
2. If it is available, use `ptb.draw` to draw in on the screen.
3. Flip at the correct time.

The first task is achieved by the method *+o_ptb.+stimuli.+visual.Movie.fetch_frame()*. If another frame of the movie is available, it loads it and returns `true`. If no more frames are available, it returns `false`. This means, we can use it in a neat `while` loop.

The second task is achieved by just calling `ptb.draw` on the *Movie* object we created.

The third task is achieved by calling `ptb.flip` using the time provided by `+o_ptb.+stimuli.+visual.Movie.next_flip_time`.

In your code, it is going to look like this:

```
while my_movie.fetch_frame()
  ptb.draw(my_movie);

  ptb.flip(my_movie.next_flip_time);
end %while
```

If you want to know more

If you want to know more and/or do more advanced things with movies, please refer to the respective section of the *reference*.

Keep in mind, that a *Movie* inherits from both *+o_ptb.+stimuli.+visual.TextureBase* and *+o_ptb.+stimuli.+visual.Base*. So you can move it, scale it, add a gaussian blur and so on and so forth. . . .

How to do Tactile Stimulation

The `o_ptb` supports tactile stimulation using corticalmetrics devices. The general pattern is very similar to sending triggers and sounds.

Note: In order to use the corticalmetrics devices, you need to obtain the `CM.dll`. You should have received it from corticalmetrics.

If you work at Salzburg, you can also ask Thomas.

In order to use the device(s), you need to:

1. Connect it to the stimulation PC via USB.
2. Connect the trigger in BNC port to one of the Trigger channels of your acquisition setup.

Our experience tells us that the device is very accurate at starting the stimulation when it receives a trigger. So we exploit this property by having `o_ptb` send a trigger when you want to start the tactile stimulation. The trigger gets recorded by the EEG/MEG and the device starts stimulating at the same time.

So, we need to tell `o_ptb` three things:

1. Where is the `CM.dll`. (See above)
2. What is the Serial Number of the stimulator we want to use. (This enables us to use more than one).
3. What trigger port is the stimulator connected to.

In order to facilitate handling multiple devices, we give it a name, too. ('left') in this example.

We do this during the configuration process:

```
ptb_cfg = o_ptb.PTB_Config();

ptb_cfg.corticalmetrics_config.cm_dll = fullfile('C:\Users\exp\Documents\cm', 'CM.dll');
ptb_cfg.corticalmetrics_config.stimulator_mapping('left') = 'CM6-
↪XXXXXXXXXXXXXXXXXXXXXXXXX';
ptb_cfg.corticalmetrics_config.trigger_mapping('left') = 128;
```

We need to initialize the respective subsystems. **The trigger subsystem must be initialized first!**

```
ptb.setup_trigger;
ptb.setup_tactile;
```

Now we can get tactile stimulation objects using *+o_ptb.+stimuli.+tactile.Base*

As you can see in the reference documentation, the class takes up to 6 parameters, of which only the last is optional. We need to specify:

1. What stimulator to use.
2. What finger to stimulate.
3. At what amplitude to stimulate.
4. At what frequency to stimulate.
5. For how long.

Optionally, we can also set the phase of the stimulation oscillation.

So, to get a tactile stimulation object, stimulating the fourth finger at full intensity and 30Hz for 1 second, this is what we need to do:

```
tactile_stim = o_ptb.stimuli.tactile.Base('left', 4, 256, 30, 1);
```

And then it is just the usual pattern of *prepare* and *schedule*:

```
ptb.prepare_tactile(tactile_stim);
ptb.schedule_tactile();

ptb.play_without_flip();
```

Using the Eyetracker

o_ptb currently supports controlling the TrackPixx eyetracker. For convenient development, a dummy eyetracker is also provided that simulates calibration and otherwise just writes messages to the console.

How to initialize and use the eyetracker

No extra configuration is required if you want to use the eyetracker. However, you need to initialize as every other subsystem:

```
ptb.setup_eyetracker();
```

As at least the eye-position verification and the calibration process need to use the screen, you also need to call *ptb.setup_screen*:

```
ptb.setup_eyetracker();
ptb.setup_screen();
```

Note: Please be aware that calibrating the eye tracker only makes sense if the screen is set to fullscreen!

The next step is to verify the positions of the eyes in the camera of the eye tracker. The TrackPixx system also requires you to tell it, where the left and the right eye is. Just follow the instructions on the screen.

```
ptb.eyetracker_verify_eye_positions();
```

Next, we need to calibrate the eye tracker:

```
ptb.eyetracker_calibrate()
```

Data acquisition and forwarding of the eye positions to the MEG is started by:

```
ptb.start_eyetracker();
```

When the experiment has finished, stop the eyetracker using:

```
ptb.stop_eyetracker();
```

And save the acquired data:

```
ptb.save_eyetracker_data('eyetracker_data.mat');
```

How to create your own stimuli

Do you have to be able to do this to use o_ptb? **NO!** You can just use normal `Screen()` commands for your visual stimuli and use the `+o_ptb.stimuli.auditory.FromMatrix` audio stimulus.

However, the o_ptb base classes for these stimuli come with some advantages. For instance, you can automatically scale and move your visual stimuli. And it is not that hard!

Let's create a rectangle that we can move and scale

So, we want to create a new class that shows a filled rectangle. It should appear at the center of the screen by default and let us define the initial size and color.

So, create a new class called "Rectangle" by doing a right-click in the "Current Folder" section and choose "New File -> Class".

You will start out with something like this:

```
classdef Rectangle
  %RECTANGLE Summary of this class goes here
  %   Detailed explanation goes here

  properties
  end

  methods
```

(continues on next page)

(continued from previous page)

```
  end

end
```

The first thing we will do is to make our class inherit from the base class of all visual stimuli:

```
classdef Rectangle < o_ptb.stimuli.visual.Base
  %RECTANGLE Summary of this class goes here
  %   Detailed explanation goes here

  properties
  end

  methods
  end

end
```

Very good. Now your class is officially a visual stimulus! Now we need to teach it, what it needs to do when you want it to draw something. The `+o_ptb.stimuli.visual.Base` class defines a method called `on_draw(obj, ptb)`. This method gets called whenever o_ptb wants the class to draw something on the screen. So, we need to implement that:

```
classdef Rectangle < o_ptb.stimuli.visual.Base
  %RECTANGLE Summary of this class goes here
  %   Detailed explanation goes here

  properties
  end

  methods
    function on_draw(obj, ptb)
      ptb.screen('FillRect', [0 0 0], CenterRect([0 0 300 300], ptb.win_rect));
    end %function
  end

end
```

You can try this out now:

```
my_rect = Rectangle
ptb.draw(my_rect);
ptb.flip
```

And you see that a black rectangle is displayed at the center of the screen.

But we want the class to be more flexible. The next step would be to get the hard coded colors out of the `on_draw` methods and use properties instead:

```
classdef Rectangle < o_ptb.stimuli.visual.Base
  %RECTANGLE Summary of this class goes here
  %   Detailed explanation goes here

  properties
```

(continues on next page)

(continued from previous page)

```
    width = 300;
    height = 300;
    color = [0 0 0];
  end %properties

  methods
    function on_draw(obj, ptb)
      ptb.screen('FillRect', obj.color, CenterRect([0 0 obj.width obj.height], ptb.win_
↪rect));
    end %function
  end

end
```

This class does the same as the old version. But we defined the width, height and color as properties.

Still, we cannot choose from outside of the class, how big we want the rectangle to be and what color we want. So, we add a constructor method:

```
classdef Rectangle < o_ptb.stimuli.visual.Base
  %RECTANGLE Summary of this class goes here
  %   Detailed explanation goes here

  properties
    width;
    height;
    color;
  end %properties

  methods
    function obj = Rectangle_other(width, height, color)
      obj@o_ptb.stimuli.visual.Base();

      obj.width = width;
      obj.height = height;
      obj.color = color;
    end %function

    function on_draw(obj, ptb)
      ptb.screen('FillRect', obj.color, CenterRect([0 0 obj.width obj.height], ptb.win_
↪rect));
    end %function
  end

end
```

You might wonder what the first line in the constructor means? (`obj@o_ptb.stimuli.visual.Base();`). Remember that we inherit from another class. And that class also has a constructor that needs to be called. This line does that for you.

The rest is pretty straight forward. Our constructor takes three arguments and we assign it to the properties of the class.

We can now create and display our rectangle like this:

```
my_rect = Rectangle(200, 200, [0 0 0]);
ptb.draw(my_rect);
ptb.flip
```

Now for the last-but-one step: The +o_ptb.stimuli.visual.Base class defines two very handy methods: scale and move. In order for these to work, we need to make use of the +o_ptb.stimuli.visual.Base classe's property called destination_rect. This holds the destination rectangle where our stimulus should appear. The scale and move method recalculate its coordinates.

Here is how it works now:

```
classdef Rectangle < o_ptb.stimuli.visual.Base
  %RECTANGLE Summary of this class goes here
  %   Detailed explanation goes here

  properties
    width;
    height;
    color;
  end %properties

  methods
    function obj = Rectangle(width, height, color)
      obj@o_ptb.stimuli.visual.Base();

      ptb = o_ptb.PTB.get_instance;

      obj.width = width;
      obj.height = height;
      obj.color = color;
      obj.destination_rect = CenterRect([0 0 obj.width obj.height], ptb.win_rect);
    end %function

    function on_draw(obj, ptb)
      ptb.screen('FillRect', obj.color, obj.destination_rect);
    end %function
  end

end
```

Now check this out:

```
my_rect = Rectangle(200, 150, [0 0 0]);

for i = 1:200
  my_rect.move(1, 1);

  ptb.draw(my_rect);
  ptb.flip();
end %for
```

There is only one problem remaining: The three properties can be modified by anyone. This would lead to unexpected behavior, because I would expect that the width of the object changes if I change the width property. The easies solution is to just prohibit these properties to be read and modified from outside the class:

```
classdef Rectangle < o_ptb.stimuli.visual.Base
  %RECTANGLE Summary of this class goes here
  %   Detailed explanation goes here

  properties (Access=protected)
    width;
    height;
    color;
  end %properties

  methods
    function obj = Rectangle(width, height, color)
      obj@o_ptb.stimuli.visual.Base();

      ptb = o_ptb.PTB.get_instance;

      obj.width = width;
      obj.height = height;
      obj.color = color;
      obj.destination_rect = CenterRect([0 0 obj.width obj.height], ptb.win_rect);
    end %function

    function on_draw(obj, ptb)
      ptb.screen('FillRect', obj.color, obj.destination_rect);
    end %function
  end

end
```

That's it!

2.3 Reference

2.3.1 PTB

class +o_ptb.**PTB**(*ptb_config*)

PTB is the main class, controlling the underlying PsychToolbox and devices.

Initializing an experiment with this library normally involves the following steps:

1. Issue 'restoredefaultpath' to make sure that you work with a clean path
2. Add the top-level folder of this library to the Matlab path. You do not need to add all the subfolders!
3. Create an instance of *+o_ptb.PTB_Config* and set all the preferences you need.
4. Create an instance of this class supplying your *+o_ptb.PTB_Config* instance
5. Setup the subsystems by calling the appropriate setup methods.

IMPORTANT: Per default, your resolution is ALWAYS 1920x1080!!! If your window or screen is smaller, the content gets scaled accordingly.

Variables

- **win_handle** (PTB window handle) – The window handle of the PTB window

- **win_rect** (PTB Rect) – The window rect (i.e. the position and size) of the PTB window
- **flip_interval** (float) – The amount of time between two screen refreshes
- **width_pixel** (int) – The width of the PTB window in pixels
- **height_pixel** (int) – The height of the PTB window in pixels
- **using_datapixx_video** (bool) – True if a Datapixx system was found
- **base_path** (string) – The folder of the o_ptb.
- **assets_path** (string) – The folder of the o_ptb assets (pictures etc).

The constructor method:

static get_instance(*ptb_config*)

Return an instance of the *PTB* class.

If *ptb_config* is supplied, this methods attemps to create a new instance of *+o_ptb.PTB*. If such an instance has already been created, this method is going to fail.

If no parameter is supplied, this method returns the currently active *+o_ptb.PTB*. If none has been created earlier, it will fail.

Parameters
ptb_config (*+o_ptb.PTB_Config*, optional) – The configuration. Only supply if you want to create a new instance!

Returns
+o_ptb.PTB – The current or new *PTB*.

Methods to set up the different subsystems:

setup_screen()

Set up the screen for the experiment.

You need to call this method before doing any drawing on the screen or working with visual stimuli.

setup_audio()

Set up the audio system for the experiment.

You need to call this method before working with auditory stimuli.

setup_trigger()

Set up the triggering system for the experiment.

You need to call this method before working with triggers.

setup_response()

Set up the response system for the experiment.

You need to call this method before working with responses.

setup_tactile()

Set up the tactile system for the experiment.

You need to call this method before working with tactile stimuli.

setup_eyetracker()

Set up the eyetracker system for the experiment.

You need to call this method before working with the eyetracker.

Methods to schedule stimuli:

draw(*stimulus*)

Draw a visual stimulus.

Draw a visual stimulus to the offscreen buffer. It does not appear immediately but at the next screen flip.

Parameters

stimulus (Instance of subclass of *+o_ptb.+stimuli.+visual.Base*) – The stimulus to draw.

prepare_audio(*stimulus*, *delay*, *hold*, *mix*)

Prepare and upload an auditory stimulus.

This is the first step of using an auditory stimulus. This method will prepare the audio stimulus and upload it to the audio subsystem. This may take some processing time so it should be done when timing is not crucial. The second step would be to call *schedule_audio*.

Note: If you prepare two stimuli at overlapping time-intervals, the latter overwrites the earlier one.

Parameters

- **stimulus** (Instance of subclass of *+o_ptb.+stimuli.+auditory.Base*) – The stimulus to prepare.
- **delay** (*float, optional*) – The delay in seconds. Default = 0
- **hold** (*bool, optional*) – If this is set to true, the previous stimuli will be retained. If it is set to false, the previous stimuli will be deleted. This is useful if you want to schedule multiple sounds with different delays. Please be aware that if sounds overlap, the new one wins.
- **mix** (*bool, optional*) – If this is set to true, a new stimulus overlapping a previously prepared one will not overwrite the former. Instead, both streams will be mixed.

schedule_audio()

Schedule the prepared auditory stimuli.

This is the second step of using an auditory stimuli. This method will do the final preparations of the audio subsystem to emit the stimulus. In order to actually fire the stimulus, you need to call either *play_without_flip* to stimulate immediately or *play_on_flip* to stimulate at the next flip command.

Note: Calling this method without calling *prepare_audio* first results in the previously prepared stimulus being played again.

---

set_audio_background(*background_object*)

Set the audio background.

The sound data in background_object will be played continuously. If an auditory stimulus is prepared, it will be added to the background noise.

The background should be a signal that does not create distortions when the position of the current audio frame jumps.

The background audio will start right away.

Parameters
background_object (Instance of subclass of *+o_ptb.+stimuli.+auditory.Base*) – The sound object to play in the background

`stop_audio_background()`

Stop the background audio.

`prune_audio(seconds)`

Make sure that the audio stream is at most seconds long.

This method works on the audio data that has been prepared but not yet scheduled.

Parameters
seconds (*float*) – Maximum length of the audio stream.

`prepare_trigger(value, delay, hold)`

Prepare to fire a trigger.

This is the first step of firing a trigger. This method will upload the trigger value to the trigger subsystem. This may take some processing time so it should be done when timing is not crucial. The second step would be to call *schedule_trigger*

Parameters

- **value** (*int*) – The trigger value to prepare.
- **delay** (*float, optional*) – The delay in seconds.
- **hold** (*bool, optional*) – If this is set to true, the previous triggers will be retained. If it is set to false, the previous triggers will be deleted. This is useful if you want to schedule multiple triggers with different delays. Please be aware that if triggers overlap, the new one wins.

`schedule_trigger()`

Schedule the prepared trigger.

This is the second step of firing a trigger. This method will do the final preparations of the trigger subsystem to emit the trigger. In order to actually fire the stimulus, you need to call either *play_without_flip* to stimulate immediately or *play_on_flip* to stimulate at the next flip command.

Note: Calling this method without calling *prepare_trigger* first results in the previously prepared trigger being fired again.

`prepare_tactile(stim, delay, hold)`

Prepare a tactile stimulus.

This is the first step of doing tactile stimulation. This method prepares the stimulus, uploads it to the stimulator and prepares the triggers. This may take some processing time so it should be done when timing is not crucial. The second step would be to call *schedule_tactile*

Parameters

- **stim** (*+o_ptb.+stimuli.+tactile.Base*) – The stimulus to prepare.
- **delay** (*float, optional*) – The delay in seconds.

- **hold** (*bool, optional*) – If this is set to true, the previous stimuli will be retained. If it is set to false, the previous stimuli will be deleted. This is useful if you want to schedule multiple stimuli with different delays. Please be aware that if stimuli overlap, the new one wins.

schedule_tactile()

Schedule the prepared tactile stimuli.

This is the second step of doing tactile stimulation. This method will do the final preparations of the tactile subsystem to emit the stimuli. In order to actually fire the stimulus, you need to call either *play_without_flip* to stimulate immediately or *play_on_flip* to stimulate at the next flip command.

Note: Calling this method without calling *prepare_trigger* first results in the previously prepared trigger being fired again.

Methods to emit stimuli:

play_without_flip()

Play scheduled sounds and fire triggers immediately.

play_on_flip()

Play scheduled sounds and fire triggers at the next flip.

Please note that in order for this to work, you need to use the *flip* shortcut function provided by this class.

flip(*varargin*)

Issues a screen flip.

This is basically a shortcut to the PTB Screen('Flip', ...) command that also takes sure that stimuli scheduled with *play_on_flip* are fired. So, always use this one.

Parameters and return values are identical to the Screen('Flip') command with the exception that you must not provide the windowPtr.

Methods to handle responses:

wait_for_keys(*keys*, *until*)

Wait for specified keys or buttons to be pressed.

This method uses the underlying response subsystem to wait keys or buttons being pressed by the participant. It uses the button mapping specified in *+o_ptb.PTB_Config*

Parameters

- **keys** (*cell array of key_ids*) – A cell array of key_ids to wait for.
- **until** (*float, optional*) – Timeout of the method in PTB seconds.

Returns

- **keys_pressed** (*cell array of strings*) – A cell array of the keys that were pressed. Empty if no key was pressed.
- **timestamp** (*cell array of floats*) – The timestamps of the key presses.

start_record_keys()

Starts recording button presses.

As soon as you call this method, button presses are recorded internally. After calling *stop_record_keys*, you can query what keys (if any) have been pressed in that interval using *get_recorded_keys*,

stop_record_keys()

Stops recording button presses.

As soon as you call this method, you can query what keys (if any) have been pressed in the interval between the call to *start_record_keys* and this call using *get_recorded_keys*.

get_recorded_keys(*keys*)

Return recorded keys.

Return keys pressed between *start_record_keys* and *stop_record_keys*

Returns

- **keys_pressed** (*cell array of strings*) – A cell array of the keys that were pressed. Empty if no key was pressed.
- **timestamp** (*cell array of floats*) – The timestamps of the key presses.

Methods to control the Eyetracker

eyetracker_verify_eye_positions()

Verify the position of the eyes for the eyetracker.

eyetracker_calibrate(*out_folder*)

Do eyetracker calibration

Parameters

out_folder (*str*) – Folder where to store the logs and images of the calibration.

start_eyetracker()

Start the eyetracker.

stop_eyetracker()

Stop the eyetracker

save_eyetracker_data(*f_name*)

Save the data acquired during an eyetracker measurement run.

Parameters

fname (*str, optional*) – Filename to save the digital eyetracker data to

Other methods:

deinit()

Uninitializes the system.

Closes the window and all connections to the subsystems.

is_screen_ready()

Query if we have a valid and open window.

Returns

bool – `true` if the screen is ready.

screen(*command*, *varargin*)

Shortcut for the PTB Screen functions.

It allows you to call all the PTB Screen function that need a windowPtr as their first parameter. Instead of providing it yourself, it gets inserted automatically. The rest of the parameters and return values is exactly the same.

screenshot(*fname*)

Takes a screenshot of the current display and save it to a file.

Parameters
fname (*string*) – The filename to save the screenshot to.

wait_for_stimulators()

Wait until the tactile stimulators are ready.

static is_lab_pc()

Check if the current PC is a lab stimulus PC.

Whether it is a lab PC is determined by the *O_PTB_IS_LAB_PC* environment variable.

Returns
bool – `true` if the this is a lab stimulus PC.

2.3.2 PTB_Config

PTB_Config is the main configuration class for `o_ptb`. In your code, you would create an instance of this class like this:

```
ptb_cfg = o_ptb.PTB_Config();
```

Then you can use ptb_cfg like a structure with the only difference that all the fields are already in there. If, for instance, you want to run the experiment in window mode, you need to do:

```
ptb_cfg.fullscreen = false;
ptb_cfg.window_scale = 0.2;
```

PTB_Config only contains only a small subset of the available options. The configuration of the subsystems is done via subconfigs.

- *Basic Configuration*
- *Audio Configuration*
- *Trigger Configuration*
- *Response Configuration*
- *Eyetracker Configuration*
- *Tactile Configuration*
- *Defaults for certain stimuli*
- *o_ptb Internals*

Basic Configuration

class +o_ptb.**PTB_Config**

Main configuration class for o_ptb.

Some configuration options for subsystems (audio and response) are in subconfigs. Some advanced options can be found in the internal_config field.

Variables

- **fullscreen** (bool) – Set to true if you want the experiment run in fullscreen mode. If you set it to false, you also need to specifiy window_size. Default: true.
- **window_scale** (float) – The scale of the window when fullscreen = false is specified. Default: 1.
- **draw_borders_sbg** (bool) – Draws a black border around the screen. Only needed in Salzburg. (Probably) Default: true.
- **flip_horizontal** (bool) – Whether to flip the output left/right. Default: true.
- **hide_mouse** (bool) – Whether to hide the mouse cursor. Default: true.
- **background_color** (int or array of int) – The background color. Accepts all values accepted by PTB. Default: +o_ptb.+constants.PTB_Colors.grey.
- **skip_sync_test** (bool) – Whether to skip PTB's sync test. Default: false.
- **force_datapixx** (bool) – If set to true, the experiment will stop with an error if no Datapixx was found. Default: false.
- **disable_datapixx** (bool) – If set to true, o_ptb will assume that no datapixx system is present. Default: false.
- **crappy_screen** (bool) – If set to true, PTB will ignore if it thinks your video setup is super crappy. Don't use this setting when really running an experiment! Default: false.
- **force_real_triggers** (bool) – If set to true, the experiment will not run if no hardware based triggering system has been detected. Default: false.
- **audio_system_cutoff** (float) – Cutoff frequency of the audio system in Hz. Useful for tube systems used in MEGs. This value is normally only used to fix the computation of loudness values of sounds like *lufs* and *RMS* and has no effect on the actual playback. If you want to simulate the effect of the tube system using your soundcard, see +o_ptb.PTB_subconfigs.PsychPortAudio.simulate_audio_cutoff. Defaults to 0 or the value of *O_PTB_AUDIO_CUTOFF*
- **psychportaudio_config** (*+o_ptb.+PTB_subconfigs.PsychPortAudio*) – Subconfig for the Psychportaudio sound system.
- **datapixxaudio_config** (*+o_ptb.+PTB_subconfigs.DatapixxAudio*) – Subconfig for the Datapixx sound system.
- **labjacktrigger_config** (*+o_ptb.+PTB_subconfigs.LabjackTriggers*) – Subconfig for the Labjack trigger system.
- **lsltrigger_config** (*+o_ptb.+PTB_subconfigs.LSLTriggers*) – Subconfig for triggering with LabStreamingLayer.
- **keyboardresponse_config** (*+o_ptb.+PTB_subconfigs.KeyboardResponse*) – Subconfig for the Keyboard response system.
- **datapixxresponse_config** (*+o_ptb.+PTB_subconfigs.DatapixxResponse*) – Subconfig for the Datapixx response system.

- **corticalmetrics_config** (*+o_ptb.+PTB_subconfigs.CorticalMetricTactile*) – Subconfig for the CorticalMetrics tactile stimulation system.
- **internal_config** (*+o_ptb.+PTB_subconfigs.PTBInternal*) – Subconfig for advanced options.

real_experiment_sbg_cdk(*do_it*)

Shortcut for real experiment configuration.

Sets all settings to the correct state for a real experiment run. If you have a function that does the configuration for you, some of those are going to be in "debug" mode most of the time. This method sets them all to values needed to get reliable timings etc.

Parameters
do_it (*bool*) – If set to `true` settings are changed.

real_experiment_settings()

Check whether all settings are in "real experiment" mode.

Returns
bool – `true` if all settings are in "real experiment" mode.

Audio Configuration

class +o_ptb.+PTB_subconfigs.**DatapixxAudio**

Configuration for the Datapixx audio subsystem.

The values all have sensible defaults. Normally, you do not need to change something here.

Variables

- **freq** (`float`) – The sampling frequency. Default: `96000`.
- **volume** (`float or array of floats`) – The volume of the sounds. If it is only one number, the volume is set for both the participant's headphones and the outside loudspeakers. If it is a vector with two elements, the first indicates the volume at the participants while the second indicates the volume at the loudspeakers at the stimulation computer. Default: `0.5`.
- **buffer_address** (`int`) – The memory address of the audio buffer within the Datapixx. If you do not know, what this means, do not touch it! Default: 90e6

class +o_ptb.+PTB_subconfigs.**PsychPortAudio**

Configuration of the PsychportAudio audio subsystem.

The values all have sensible defaults. Normally, you do not need to change something here.

Variables

- **freq** (`int`) – The sampling frequency. Default: The highest possible or the value of *O_PTB_PSYCHPORTAUDIO_SFREQ*.
- **mode** (`int`) – The mode of operations. 1 means only ouput. Default: `1`.
- **reqlatencyclass** (`int`) – How much the system tries to give you low latencies. Just use the default. (`4`).
- **device** (`int`) – The device to use. Defaults to the first available one or the value of *O_PTB_PSYCHPORTAUDIO_DEVICE*.
- **simulate_audio_cutoff** (`bool`) – If *true*, simulate the audio cutoff.

Trigger Configuration

class +o_ptb.+PTB_subconfigs.**LabjackTriggers**

Configuration for the Labjack Triggering System.

Please note that you need a recent version of python to use the Labjack!

Sensible defaults are provided for all of them, so normally you do not need to change them.

Variables

- **channel_group** (+labjack.Labjack.ChannelGroup) – The Labjack U3 (the only one supported right now) has three different channel groups. The relevant ones for you are:
  - +labjack.Labjack.ChannelGroup.EIO: If you have the plug attached.
  - +labjack.Labjack.ChannelGroup.FIO: If you use the wires.
- **method** (+labjack.Labjack.TriggerMethod) – There are three different methods to do the triggering: * +labjack.Labjack.TriggerMethod.MULTI: Uses

  all 8 channels of the chosen channelgroup. Values are binary coded. This is the normal mode of operation you should be familar with.

  - +labjack.Labjack.TriggerMethod.SINGLE: Uses only one bit to signal a trigger. Use the single_channel property to configure which one you want. Obviously, it is impossible to signal different values this way.
  - +labjack.Labjack.TriggerMethod.PULSEWIDTH: Uses "Pulse Width Modulation" to signal different bit values using only one channel. This is done by modulating the time, the trigger line stays up or down.
    * 10ms = 1.
    * 20ms = 0.

    The num_bits property sets the number of bits that are coded. Example: num_bits = 5 and you wan to send the trigger value 5. 5 in binary representation with 5 bits is: 00101. This would be coded like this:

    1. trigger up for 20ms
    2. trigger down for 20ms
    3. trigger up for 10ms
    4. trigger down for 20ms
    5. trigger up for 10ms
- **single_channel** (int) – The channel to use for single or pulsewidth triggering.
- **num_bits** (int) – The number of bits to code when using pulsewidth mode. Must be uneven.

class +o_ptb.+PTB_subconfigs.**LSLTriggers**

Configuration options for LabStreamingLayer based triggering.

Please be aware that you need to explicitly request using this system because it is impossible to automatically determine whether it should be used as it is not hardware based.

You also need to have a current version of liblsl-Matlab (https://github.com/labstreaminglayer/liblsl-Matlab) that you also need to configure and compile the mex files for your system.

Variables

- **liblsl_matlab_path** (string) – The full path to liblsl-Matlab.
- **trigger_type** (string) – Triggers can be sent as strings like <MARKER>#code</MARKER> or as 32bit integers. If you want strings, set this to string, otherwise to int. Default: string.
- **stream_id** (string) – ID of the LSL stream. Default: o_ptb_marker_stream.

Response Configuration

class +o_ptb.+PTB_subconfigs.**DatapixxResponse**

Configuration for the Datapixx response system.

The only thing to do here is setup the mapping between the response names and the actual buttons.

Suppose, we have two possible reponses yes and no that we want to map to the blue and red button of the response box, this is how it is done:

```
ptb_config.datapixxresponse_config.button_mapping('yes') = ptb_config.
↪datapixxresponse_config.Blue;
ptb_config.datapixxresponse_config.button_mapping('no') = ptb_config.
↪datapixxresponse_config.Red;
```

For further details, please take a look at *How to get responses*.

class +o_ptb.+PTB_subconfigs.**KeyboardResponse**

Configuration for the Keyboard response system.

The only thing to do here is setup the mapping between the response names and the actual keys.

If necessary, you can also set the device number of the keyboard you want to use.

Suppose, we have two possible reponses yes and no that we want to map to the blue and red button of the response box, this is how it is done:

```
ptb_config.datapixxresponse_config.button_mapping('yes') = KbName('Right');
ptb_config.datapixxresponse_config.button_mapping('no') = KbName('Left');
```

For further details, please take a look at *How to get responses*.

Variables

device_number (int or []) – The device number of the Keyboard. Default: [].

Eyetracker Configuration

class +o_ptb.+PTB_subconfigs.**DatapixxTrackPixx**

Configuration for the Datapixx Eyetracker System

The values all have sensible defaults. Normally, you do not need to change something here.

Variables

- **lens** (int) – The lens type. Default: 1. Possible values are:
  - 0: 25mm
  - 1: 50mm
  - 2: 75mm

- **distance** (int) – Distance between the eye and the eyetrackerin cm. Default: 82
- **analogue_eye** (int) – Which eye to use for analogue output. Default: 0. Possible values are:
  - 0: left
  - 1: right
- **led_intensity** (int) – Intensity of the infrared LED lights. Must be between 0-8. High values work well for bright eyes while low values seem to work better for dark eyes. Default: 8
- **buffer_address** (int) – The memory address of the eyetracker buffer within the Datapixx. If you do not know, what this means, do not touch it! Default: 12e6

Tactile Configuration

class +o_ptb.+PTB_subconfigs.**CorticalMetricTactile**

Configuration for the CorticalMetrics Tactile Stimulator.

See *How to do Tactile Stimulation* for details how to use it.

Variables

- **cm_dll** (string) – Full path to the CM.dll.
- **stimulator_mapping** (containers.Map) – Maps an arbitrary name to the serial number of the stimulator.
- **trigger_mapping** (containers.Map) – Maps the name give at stimulator_mapping to the trigger port The device is connected to.

Defaults for certain stimuli

class +o_ptb.+PTB_subconfigs.**Defaults**

Configuration of some default values for certain stimuli.

Variables

- **text_size** (float) – Default text size. Default: 46.
- **text_wrapat** (int) – Number of characters in one line. Default: 80.
- **text_vspacing** (float) – Vertical space between two lines of text. Default: 1.
- **text_color** (int or array of ints) – Text color. Default: +o_ptb.+constants.PTB_Colors.black.
- **fixcross_size** (float) – Size of the fixation cross in pixels. Default: 120.
- **fixcross_width_ratio** (float) – Ratio between the length of each line of the fixation cross and its width. Default: 0.25.

o_ptb Internals

class +o_ptb.+PTB_subconfigs.**PTBInternal**

PTBInternal provides some advanced configuration options.

Sensible defaults are provided for all of them, so normally you do not need to change them.

2.3.3 Environment Variables

Certain aspects of the o_ptb can be preconfigured by setting so-called environment variables. On Windows, you can set these in the System Settings. On Mac and Linux you do this in the /etc/environment, bashrc, zshrc or .pam_environment file.

General Configuration

O_PTB_PTB_FOLDER

The location of the Psychtoolbox.

O_PTB_USE_DECORATED_WINDOW

Use decorated windows for the PTB screen when not run in fullscreen mode.

O_PTB_IS_LAB_PC

You should set this variable to `true` on a Stimulation PC in a lab. This will lead so a warning if an experiment is run in debug mode.

O_PTB_AUDIO_CUTOFF

The audio cutoff frequency. Useful for the tube stimulation systems.

Sound Configuration

O_PTB_PSYCHPORTAUDIO_DEVICE

The PortAudio device number to use.

O_PTB_PSYCHPORTAUDIO_SFREQ

The sampling frequency of the device when using PortAudio.

2.3.4 Constants

o_ptb provides some useful constant that you can use in your code.

class +o_ptb.+constants.**PTB_Colors**

This is a collection of useful color constants.

Just type this in the command window to see what colors are available:

```
o_ptb.constants.PTB_Colors
```

If you want to use the black color, for example, you can reach it like this:

```
o_ptb.constants.PTB_Colors.black
```

class +o_ptb.+constants.**PTB_TextStyles**

This is a collection of text style constants.

Just type this in the command window to see what colors are available:

```
o_ptb.constants.PTB_TextStyles
```

If you want to use the bold style, for example, you can reach it like this:

```
o_ptb.constants.PTB_TextStyles.Bold
```

2.3.5 Stimulation Classes

Visual Stimulus Classes

- *Display Basic Items of an Experiment*
- *Display Images*
- *Display Text*
- *Display geometric shapes*
- *Display advanced stimuli*
- *Base classes*

Display Basic Items of an Experiment

class +o_ptb.+stimuli.+visual.**FixationCross**(*color*)

Draw a fixation cross.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters
color (*int or array of three ints*) – The color of the fixation cross.

Display Images

class +o_ptb.+stimuli.+visual.**Image**(*image*)

Draw an image read from a file.

This class provides all methods of *+o_ptb.+stimuli.+visual.TextureBase* and *+o_ptb.+stimuli.+visual.Base*.

Parameters
image (*string*) – The filename of the image.

Display Text

class +o_ptb.+stimuli.+visual.**Text**(*text*)

Draw formatted text.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters
text (*string*) – The text to display

Variables

- **size** (float) – The font size of the text. Defaults to 46.
- **style** (int) – The style of the text. You can use the constants defined in *+o_ptb.+constants.PTB_TextStyles*. Defaults to +o_ptb.+constants.PTB_TextStyles.Normal.
- **font** (string) – The font used for drawing the text. Defaults to Arial.
- **sx** (float) – The x coordinates of the text. Defaults to the center of the destination_rect (normally the whole window).
- **sy** (float) – The y coordinates of the text. Defaults to the center of the destination_rect (normally the whole window).
- **color** (int or array of three ints) – The color of the text. You can use the constants defined in *+o_ptb.+constants.PTB_Colors* Defaults to +o_ptb.+constants.PTB_Colors.black.
- **wrapat** (int) – If set, text will automatically be continued on the next line if one line exceeds that amount of characters.
- **vspacing** (int) – The spacing between vertical lines. If your lines overlap, increase this property. Defaults to 1.

class +o_ptb.+stimuli.+visual.**TextFile**(*f_name*)

Read formatted text from a file and draw it.

This class provides all methods of *+o_ptb.+stimuli.+visual.Text*.

Parameters
f_name (*string*) – The filename to read the text from.

Display geometric shapes

class +o_ptb.+stimuli.+visual.**Line**(*height*, *width*, *color*, *pen_width*)

Draw a line.

The line is initially centered on the screen. Use the move() method to move it to the desired location.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **height** (*float*) – The distance from the bottom to the top of the line.
- **width** (*float*) – The distance from left to right of the line.
- **color** (*int or array of three ints*) – The color of the circle.
- **pen_size** (*float*) – The width of the circle outline.

class +o_ptb.+stimuli.+visual.**FrameCircle**(*radius*, *color*, *pen_size*)

Draw the outline of a circle.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **radius** (*float*) – The radius of the circle.
- **color** (*int or array of three ints*) – The color of the circle.
- **pen_size** (*float, optional*) – The width of the circle outline.

class +o_ptb.+stimuli.+visual.**FrameOval**(*width*, *height*, *color*, *pen_size*)

Draw the outline of an oval.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **width** (*float*) – The width of the oval.
- **height** (*float*) – The height of the oval.
- **color** (*int or array of three ints*) – The color of the oval.
- **pen_size** (*float, optional*) – The width of the ovals outline.

class +o_ptb.+stimuli.+visual.**FilledCircle**(*radius*, *color*)

Draw a filled circle.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **radius** (*float*) – The radius of the circle.
- **color** (*int or array of three ints*) – The color of the circle.

class +o_ptb.+stimuli.+visual.**FilledOval**(*width*, *height*, *color*)

Draw a filled oval.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **width** (*float*) – The width of the oval.
- **height** (*float*) – The height of the oval.
- **color** (*int or array of three ints*) – The color of the oval.

class +o_ptb.+stimuli.+visual.**FrameCircleArc**(*radius*, *color*, *start_angle*, *total_angle*, *pen_size*)

Draw the outline of a circle arc.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **radius** (*float*) – The radius of the circle.
- **color** (*int or array of three ints*) – The color of the circle.
- **start_angle** (*float*) – The angle in degrees at which to start drawing.
- **total_angle** (*float*) – The angle of the arc in degrees.
- **pen_size** (*float, optional*) – The width of the circle outline.

class +o_ptb.+stimuli.+visual.**FrameOvalArc**(*width*, *height*, *color*, *start_angle*, *total_angle*, *pen_size*)

Draw the outline of a oval arc.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **width** (*float*) – The width of the oval.
- **height** (*float*) – The height of the oval.
- **color** (*int or array of three ints*) – The color of the circle.
- **start_angle** (*float*) – The angle in degrees at which to start drawing.
- **total_angle** (*float*) – The angle of the arc in degrees.
- **pen_size** (*float, optional*) – The width of the circle outline.

class +o_ptb.+stimuli.+visual.**FilledCircleArc**(*radius*, *color*, *start_angle*, *total_angle*)

Draw a filled circle arc.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **radius** (*float*) – The radius of the circle.
- **color** (*int or array of three ints*) – The color of the circle.
- **start_angle** (*float*) – The angle in degrees at which to start drawing.
- **total_angle** (*float*) – The angle of the arc in degrees.

class +o_ptb.+stimuli.+visual.**FilledOvalArc**(*width*, *height*, *color*, *start_angle*, *total_angle*)

Draw a filled oval arc.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **width** (*float*) – The width of the oval.
- **height** (*float*) – The height of the oval.
- **color** (*int or array of three ints*) – The color of the circle.
- **start_angle** (*float*) – The angle in degrees at which to start drawing.
- **total_angle** (*float*) – The angle of the arc in degrees.
- **pen_size** (*float, optional*) – The width of the circle outline.

Display advanced stimuli

class +o_ptb.+stimuli.+visual.**Gabor**(*width*, *height*)

Draw a Gabor patch.

Note: It is mandatory to set the following properties after the class is instantiated:

- frequency
- sc
- contrast

This class provides all methods of *+o_ptb.+stimuli.+visual.TextureBase* and *+o_ptb.+stimuli.+visual.Base*.

Parameters

- **width** (*float*) – The width of the gabor patch.
- **height** (*float*) – The height of the gabor patch.

Variables

- **rotate** (float) – The rotation of the gabor in degrees. Default = 0.
- **phase** (float) – The phase of the gabor in degrees. Default = 180.
- **aspectration** (float) – Leave at 1 (the default) if you want the gabor to be as wide as it is high.
- **frequency** (float) – The spatial frequency in cycles per pixel. No default.
- **sc** (float) – The sigma value of the gauss function. No default.
- **contrast** (float) – The contrast of the gabor. No default.

class +o_ptb.+stimuli.+visual.**Movie**(*f_name*)

Read a movie file and provide the frames.

Please refer to the *tutorial* for detailed instructions.

This class provides all methods of *+o_ptb.+stimuli.+visual.TextureBase* and *+o_ptb.+stimuli.+visual.Base*.

Parameters

f_name (*string*) – The filename of the movie to load.

Variables

- **duration** (float) – Duration of the movie in seconds.
- **fps** (float) – Framrate of the movie.
- **next_flip_time** (float) – Timestamp of the flip for the currently loaded frame.

fetch_frame()

Fetch the next frame of the movie.

Returns

bool – true if another frame was available.

resync()

Skip frames of the movie to catch up with current time.

start()

Start movie playback.

stop()

Stop movie playback.

Base classes

class +o_ptb.+stimuli.+visual.**Base**

This is the base class for all visual stimuli.

This means that:

1. All visual stimulus classes provide all the parameters and methods of this base class. Please refer to *How to draw stuff on the screen* for details.
2. In order to create your own visual stimulus class, you need to inherite from this base class. Please refer to *How to create your own stimuli* for details.

Variables

- **height** (float) – The height of the stimulus in pixels
- **width** (float) – The width of the stimulus in pixels

center_on_screen()

Center the stimulus on the screen.

move(*movex*, *movey*)

Move the stimulus.

Move the stimulus by the amount of pixels.

Parameters

- **movex** (*float*) – Amount of pixels to move horizontally.
- **movey** (*float*) – Amount of pixels to move vertically.

move_to(*x*, *y*)

Move stimulus to x and y coordinates.

Parameters

- **x** (*float*) – The x coordinate of the destination.
- **y** (*float*) – The y coordinate of the destination.

scale(*scalex*, *scaley*)

Scale the stimulus.

Scale the stimulus by the ratios provided via the parameters.

Parameters

- **scalex** (*float*) – Ratio to scale the stimulus horizontally.
- **scaley** (*float, optional*) – Ration to scale the stimulus vertically. If ommitted, scalex is used.

class +o_ptb.+stimuli.+visual.**TextureBase**

Base class for all texture based stimuli.

Variables
rotate (float) – Rotation of the stimulus in degrees.

This class provides all methods of *+o_ptb.+stimuli.+visual.Base*.

add_gauss_blur(*stdev*, *kernel_size*)

Add gaussian blur to the stimulus.

Blur the image with a guassian kernel.

Parameters

- **stdev** (*float*) – Standard deviation of the gaussian kernel.
- **kernel_size** (*int*) – Size of the kernel.

Audio Stimulus Classes

- *Loading and generating Sounds*
- *Base classes*

Loading and generating Sounds

class +o_ptb.+stimuli.+auditory.**Wav**(*filename*)

Audio Stimulus read from a wav file.

This class provides all methods of *+o_ptb.+stimuli.+auditory.Base*.

Parameters

filename (*string*) – The filename of the wav file to load.

class +o_ptb.+stimuli.+auditory.**Sine**(*freq*, *duration*)

Sine wave audio stimulus.

This class provides all methods of *+o_ptb.+stimuli.+auditory.Base*.

Parameters

- **freq** (*float*) – Frequency of the sine wave
- **duration** (*float*) – Duration in seconds.

class +o_ptb.+stimuli.+auditory.**WhiteNoise**(*duration*)

White Noise stimulus.

This class provides all methods of *+o_ptb.+stimuli.+auditory.Base*.

Parameters

duration (*float*) – Duration in seconds.

class +o_ptb.+stimuli.+auditory.**FromMatrix**(*sound_data*, *s_rate*)

Audio Stimulus read from a matrix.

This class provides all methods of *+o_ptb.+stimuli.+auditory.Base*.

Parameters

- **sound_data** (*matrix of floats*) – The sound data. One channel per row, one sample per column. The data must not exceed -1 / +1.
- **s_rate** (*float*) – The sampling rate of the sound data.

Base classes

class +o_ptb.+stimuli.+auditory.**Base**

This is the base class for all auditory stimuli.

This means that:

1. All auditory stimulus classes provide all the parameters and methods of this base class. Please refer to *Using Triggers and Sound* for details.
2. In order to create your own auditory stimulus class, you need to inherit from this base class.

Note: All volume-related attributes (i.e., amplification_factor, rms, db, lufs and absmax) are related to each other. If you change one of them, the others reflect the new volume.

Note: If `+o_ptb.PTB_Config.audio_system_cutoff` or *O_PTB_AUDIO_CUTOFF* is set, `rms`, `rms_db` and `lufs` the sound will be lowpass filtered at that frequency before these values are computed.

Note: It is not possible to play sounds at a volume that would lead to clipping.

Note: Sampling rate conversion is done automatically.

Variables

- **amplification_factor** (`[float float]`) – The factor by which each channel of the sound is amplified.
- **rms** (`[float float]`) – The root-mean-square of the two channels.
- **rms_db** (`[float float]`) – The RMS expressed in dB.
- **lufs** (`[float float]`) – The "Loudness Unit Full Scale" of the two channels. This is a comparable to RMS but accounts for the psychophysic properties of human hearing. It is also on a logarithmic scale like dB.
- **db** (`[float float]`) – The maximum amplitude of both channels expressed in dB. 0dB is the maximum volume.
- **absmax** (`[float float]`) – The maximum amplitude of both channels.
- **muted_channels** (`int or array of ints`) – If empty (i.e. []), both channels are played. If set to 1, the left channel is muted. If set to 2, the right channel is muted. If set to [1 2], both channels are muted.
- **duration** (`float`) – The duration of the sound in seconds.
- **n_samples** (`int`) – The number of samples of the sound.
- **angle** (`float`) – The angle of the direction where the sound comes from. This is used to calculate the Interaural Time Difference using the "Woodworth model" (Woodworth 1938). Negative values make the sound appear from the left, positive values from the right. The unit is "radians" and must be between -pi and +pi.
- **head_radius** (`float`) – Radius of the head in meters. Used to calculate the ITD.

`amplify`(*factor*)

Amplify the sound.

Parameters
factor (*float*) – Amplification factor.

`amplify_db`(*db*)

Amplify the sound by dB.

Parameters
db (*float*) – dB to add to the volume.

`amplitude_modulate`(*mod_freq*, *mod_depth*)

Apply amplitude modulation to the sound.

Parameters

- **mod_freq** (*float*) – Frequency of the modulation
- **mod_depth** (*float*) – Depth of the modulation

`apply_cos_ramp`(*duration*)

Apply a cosine ramp to the start and end of the sound.

Parameters
duration (*float*) – Duration of the ramp in seconds.

`apply_hanning`()

Apply a hanning window to the sound.

`apply_sin_ramp`(*duration*)

Apply a sine ramp to the start and end of the sound.

Parameters
duration (*float*) – Duration of the ramp in seconds.

`debug_play_now`()

Plays the stimulus right now. Do not use in real experiment

> **Warning:** This method should only be used to interactively check a stimulus. It should not be used in your real experiment script!

`fadeinout`(*fade_length*)

Apply linear fade in and fade out.

Parameters
fade_length (*float*) – Duration of the fade in seconds.

`filter_bandpass`(*l_freq*, *h_freq*, *l_transition_width*, *h_transition_width*, *max_passband_ripple*, *max_stopband_ripple*)

Apply bandpass filter.

Parameters

- **l_freq** (*float*) – Low edge frequency
- **h_freq** (*float*) – High edge frequency
- **l_transition_width** (*float, optional*) – Bandwidth between the lower passband and the stop-band. Default = l_freq * 0.05

- **h_transition_width** (*float, optional*) – Bandwidth between the higher passband and the stopband. Default = h_freq * 0.05
- **max_passband_ripple** (*float, optional*) – Maximum allowed passband ripple. Default = 3
- **max_stopband_ripple** (*float, optional*) – Maximum allowed stopband ripple. Default = 20

`filter_bandstop`(*l_freq*, *h_freq*, *l_transition_width*, *h_transition_width*, *max_passband_ripple*, *max_stopband_ripple*)

Apply bandstop filter.

Parameters

- **l_freq** (*float*) – Low edge frequency
- **h_freq** (*float*) – High edge frequency
- **l_transition_width** (*float, optional*) – Bandwidth between the lower passband and the stopband. Default = l_freq * 0.05
- **h_transition_width** (*float, optional*) – Bandwidth between the higher passband and the stopband. Default = h_freq * 0.05
- **max_passband_ripple** (*float, optional*) – Maximum allowed passband ripple. Default = 3
- **max_stopband_ripple** (*float, optional*) – Maximum allowed stopband ripple. Default = 20

`filter_highpass`(*freq*, *transition_width*, *max_passband_ripple*, *max_stopband_ripple*)

Apply highpass filter.

Parameters

- **freq** (*float*) – Edge frequency
- **transition_width** (*float, optional*) – Bandwidth between the passband and the stopband. Default = freq * 0.05
- **max_passband_ripple** (*float, optional*) – Maximum allowed passband ripple. Default = 3
- **max_stopband_ripple** (*float, optional*) – Maximum allowed stopband ripple. Default = 20

`filter_lowpass`(*freq*, *transition_width*, *max_passband_ripple*, *max_stopband_ripple*)

Apply lowpass filter.

Parameters

- **freq** (*float*) – Edge frequency
- **transition_width** (*float, optional*) – Bandwidth between the passband and the stopband. Default = freq * 0.05
- **max_passband_ripple** (*float, optional*) – Maximum allowed passband ripple. Default = 3
- **max_stopband_ripple** (*float, optional*) – Maximum allowed stopband ripple. Default = 20

`flip_polarity`()

Flip the polarity of the sound.

`flip_sound`()

Flip the sound so it will be played backwards.

plot_spectrum()

Plot spectrum

plot_waveform()

Plot waveform

plus(*a*, *b*)

Add two sounds using the + operator.

If you have two sounds sound_a and sound_b, you can add the two by doing this:

```
new_sound = sound_a + sound_b;
```

save_wav(*fname*, *srate*)

Save the sound data to a wav file.

Parameters

- **fname** (*string*) – The filename to save the data to.
- **srate** (*int, optional*) – Sampling rate. defaults to the sampling rate of the stimulus.

set_to_max_amplification()

Set this stimulus to the maximum volume without clipping.

vocode(*n_channels*, *freq_range*)

Vocode the sound.

Parameters

- **n_channels** (*int*) – Number of vocoder channels.
- **freq_range** (*[float float], optional*) – The frequency range to use for vocoding.

Tactile Classes

- *Base Classes*

Base Classes

class +o_ptb.+stimuli.+tactile.**Base**(*stimulator*, *finger*, *amplitude*, *frequency*, *duration*, *phase*)

Base class for tactile stimuli.

As opposed to all the other base classes, this one can be used directly.

For details how to do tactile stimulation, refer to the *respective tutorial*.

Parameters

- **stimulator** (*string*) – Mapped name of the stimulator to use.
- **finger** (*int*) – Finger to stimulate.
- **amplitude** (*int*) – Amplitude at which to stimulate. Range is 0 to 256.
- **frequency** (*float*) – Stimulation frequency.
- **duration** (*float*) – Duration of the stimulation in seconds.

- **phase** (*float, optional*) – Initial phase of the stimulation oscillation.

2.3.6 Tools and convenience function

- *Initialize and Setup*
- *Audio*

Initialize and Setup

+o_ptb.**init_ptb**(*ptb_folder*)

Initialize o_ptb.

Parameters

ptb_folder (*string, optional*) – Path to the Psychtoolbox folder. If this is not provided, the environment variable "O_PTB_PTB_FOLDER" will be used if set. Otherwise the current folder and all subfolder will be searched for an installation.

Audio

+o_ptb.+tools.+audio.**equalize_rms**(*stims*, *max_amp*)

Set all stimuli to the same RMS value.

All stimuli in the cell array stims will be set to the maximum RMS value of all stimuli in the array. In order to avoid clipping, no sound will be louder than max_amp.

Parameters

- **stims** (cell array of *+o_ptb.+stimuli.+auditory.Base*) – Cell array of the stimuli to equalize.
- **max_amp** (*float*) – Maximum amplitude of any sound.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

MATLAB MODULE INDEX

+

+o_ptb, 44

INDEX

Symbols

+o_ptb (*module*), 44

A

add_gauss_blur() (*+o_ptb.+stimuli.+visual.TextureBase method*), 56
amplify() (*+o_ptb.+stimuli.+auditory.Base method*), 58
amplify_db() (*+o_ptb.+stimuli.+auditory.Base method*), 59
amplitude_modulate() (*+o_ptb.+stimuli.+auditory.Base method*), 59
apply_cos_ramp() (*+o_ptb.+stimuli.+auditory.Base method*), 59
apply_hanning() (*+o_ptb.+stimuli.+auditory.Base method*), 59
apply_sin_ramp() (*+o_ptb.+stimuli.+auditory.Base method*), 59

B

Base (*class in +o_ptb.+stimuli.+auditory*), 58
Base (*class in +o_ptb.+stimuli.+tactile*), 61
Base (*class in +o_ptb.+stimuli.+visual*), 56

C

center_on_screen() (*+o_ptb.+stimuli.+visual.Base method*), 56
CorticalMetricTactile (*class in +o_ptb.+PTB_subconfigs*), 49

D

DatapixxAudio (*class in +o_ptb.+PTB_subconfigs*), 46
DatapixxResponse (*class in +o_ptb.+PTB_subconfigs*), 48
DatapixxTrackPixx (*class in +o_ptb.+PTB_subconfigs*), 48
debug_play_now() (*+o_ptb.+stimuli.+auditory.Base method*), 59
Defaults (*class in +o_ptb.+PTB_subconfigs*), 49
deinit() (*+o_ptb.PTB method*), 43
draw() (*+o_ptb.PTB method*), 40

E

environment variable
    O_PTB_AUDIO_CUTOFF, 45, 50, 58
    O_PTB_IS_LAB_PC, 50
    O_PTB_PSYCHPORTAUDIO_DEVICE, 46, 50
    O_PTB_PSYCHPORTAUDIO_SFREQ, 46, 50
    O_PTB_PTB_FOLDER, 22, 50
    O_PTB_USE_DECORATED_WINDOW, 50
equalize_rms() (*in module +o_ptb.+tools.+audio*), 62
eyetracker_calibrate() (*+o_ptb.PTB method*), 43
eyetracker_verify_eye_positions() (*+o_ptb.PTB method*), 43

F

fadeinout() (*+o_ptb.+stimuli.+auditory.Base method*), 59
fetch_frame() (*+o_ptb.+stimuli.+visual.Movie method*), 55
FilledCircle (*class in +o_ptb.+stimuli.+visual*), 53
FilledCircleArc (*class in +o_ptb.+stimuli.+visual*), 54
FilledOval (*class in +o_ptb.+stimuli.+visual*), 53
FilledOvalArc (*class in +o_ptb.+stimuli.+visual*), 54
filter_bandpass() (*+o_ptb.+stimuli.+auditory.Base method*), 59
filter_bandstop() (*+o_ptb.+stimuli.+auditory.Base method*), 60
filter_highpass() (*+o_ptb.+stimuli.+auditory.Base method*), 60
filter_lowpass() (*+o_ptb.+stimuli.+auditory.Base method*), 60
FixationCross (*class in +o_ptb.+stimuli.+visual*), 51
flip() (*+o_ptb.PTB method*), 42
flip_polarity() (*+o_ptb.+stimuli.+auditory.Base method*), 60
flip_sound() (*+o_ptb.+stimuli.+auditory.Base method*), 60
FrameCircle (*class in +o_ptb.+stimuli.+visual*), 52
FrameCircleArc (*class in +o_ptb.+stimuli.+visual*), 53
FrameOval (*class in +o_ptb.+stimuli.+visual*), 53

FrameOvalArc (*class in +o_ptb.+stimuli.+visual*), 53
FromMatrix (*class in +o_ptb.+stimuli.+auditory*), 57

G

Gabor (*class in +o_ptb.+stimuli.+visual*), 54
get_instance() (*+o_ptb.PTB static method*), 39
get_recorded_keys() (*+o_ptb.PTB method*), 43

I

Image (*class in +o_ptb.+stimuli.+visual*), 51
init_ptb() (*in module +o_ptb*), 62
is_lab_pc() (*+o_ptb.PTB static method*), 44
is_screen_ready() (*+o_ptb.PTB method*), 43

K

KeyboardResponse (*class in +o_ptb.+PTB_subconfigs*), 48

L

LabjackTriggers (*class in +o_ptb.+PTB_subconfigs*), 47
Line (*class in +o_ptb.+stimuli.+visual*), 52
LSLTriggers (*class in +o_ptb.+PTB_subconfigs*), 47

M

move() (*+o_ptb.+stimuli.+visual.Base method*), 56
move_to() (*+o_ptb.+stimuli.+visual.Base method*), 56
Movie (*class in +o_ptb.+stimuli.+visual*), 55

O

O_PTB_AUDIO_CUTOFF, 45, 58
O_PTB_PSYCHPORTAUDIO_DEVICE, 46
O_PTB_PSYCHPORTAUDIO_SFREQ, 46
O_PTB_PTB_FOLDER, 22

P

play_on_flip() (*+o_ptb.PTB method*), 42
play_without_flip() (*+o_ptb.PTB method*), 42
plot_spectrum() (*+o_ptb.+stimuli.+auditory.Base method*), 60
plot_waveform() (*+o_ptb.+stimuli.+auditory.Base method*), 61
plus() (*+o_ptb.+stimuli.+auditory.Base method*), 61
prepare_audio() (*+o_ptb.PTB method*), 40
prepare_tactile() (*+o_ptb.PTB method*), 41
prepare_trigger() (*+o_ptb.PTB method*), 41
prune_audio() (*+o_ptb.PTB method*), 41
PsychPortAudio (*class in +o_ptb.+PTB_subconfigs*), 46
PTB (*class in +o_ptb*), 38
PTB_Colors (*class in +o_ptb.+constants*), 50
PTB_Config (*class in +o_ptb*), 45
PTB_TextStyles (*class in +o_ptb.+constants*), 50
PTBInternal (*class in +o_ptb.+PTB_subconfigs*), 50

R

real_experiment_sbg_cdk() (*+o_ptb.PTB_Config method*), 46
real_experiment_settings() (*+o_ptb.PTB_Config method*), 46
resync() (*+o_ptb.+stimuli.+visual.Movie method*), 55

S

save_eyetracker_data() (*+o_ptb.PTB method*), 43
save_wav() (*+o_ptb.+stimuli.+auditory.Base method*), 61
scale() (*+o_ptb.+stimuli.+visual.Base method*), 56
schedule_audio() (*+o_ptb.PTB method*), 40
schedule_tactile() (*+o_ptb.PTB method*), 42
schedule_trigger() (*+o_ptb.PTB method*), 41
screen() (*+o_ptb.PTB method*), 43
screenshot() (*+o_ptb.PTB method*), 44
set_audio_background() (*+o_ptb.PTB method*), 40
set_to_max_amplification() (*+o_ptb.+stimuli.+auditory.Base method*), 61
setup_audio() (*+o_ptb.PTB method*), 39
setup_eyetracker() (*+o_ptb.PTB method*), 39
setup_response() (*+o_ptb.PTB method*), 39
setup_screen() (*+o_ptb.PTB method*), 39
setup_tactile() (*+o_ptb.PTB method*), 39
setup_trigger() (*+o_ptb.PTB method*), 39
Sine (*class in +o_ptb.+stimuli.+auditory*), 57
start() (*+o_ptb.+stimuli.+visual.Movie method*), 55
start_eyetracker() (*+o_ptb.PTB method*), 43
start_record_keys() (*+o_ptb.PTB method*), 42
stop() (*+o_ptb.+stimuli.+visual.Movie method*), 55
stop_audio_background() (*+o_ptb.PTB method*), 41
stop_eyetracker() (*+o_ptb.PTB method*), 43
stop_record_keys() (*+o_ptb.PTB method*), 43

T

Text (*class in +o_ptb.+stimuli.+visual*), 52
TextFile (*class in +o_ptb.+stimuli.+visual*), 52
TextureBase (*class in +o_ptb.+stimuli.+visual*), 56

V

vocode() (*+o_ptb.+stimuli.+auditory.Base method*), 61

W

wait_for_keys() (*+o_ptb.PTB method*), 42
wait_for_stimulators() (*+o_ptb.PTB method*), 44
Wav (*class in +o_ptb.+stimuli.+auditory*), 57
WhiteNoise (*class in +o_ptb.+stimuli.+auditory*), 57