

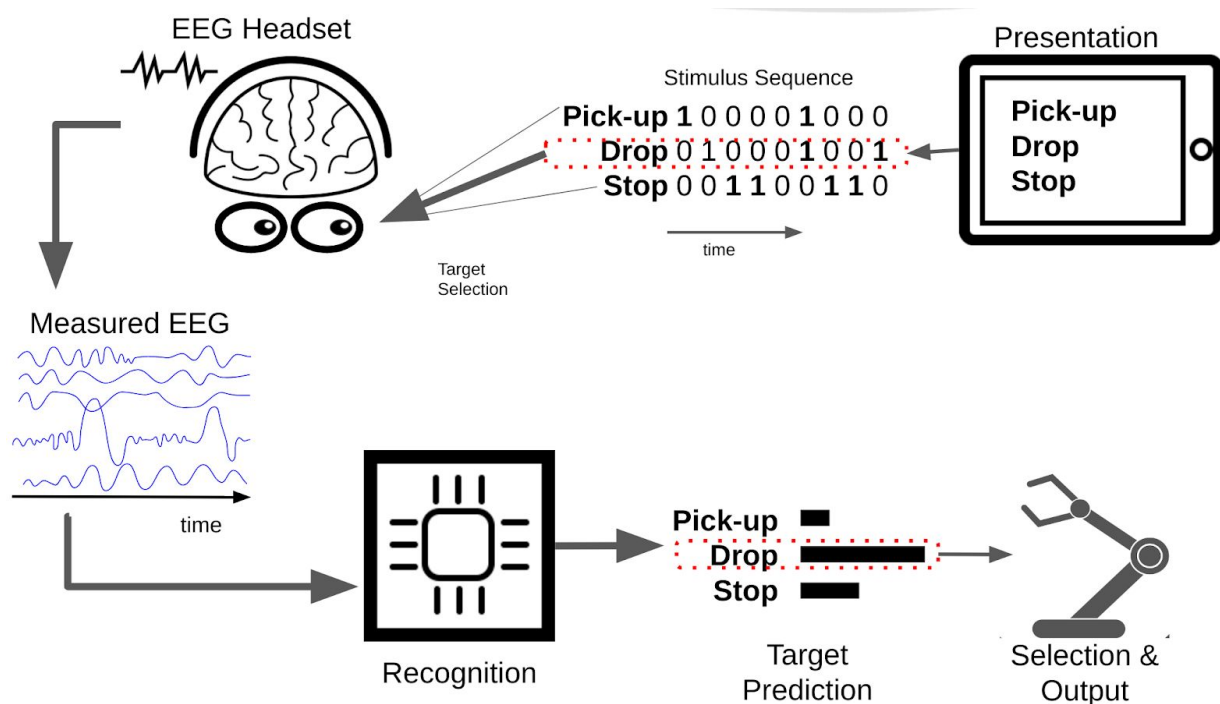
# Tutorial Example: How to control a device with a BCI

## Introduction

Utopia is a framework for making Brain Computer Interfaces developed by [MindAffect](#) B.V. This tutorial goes step-by-step through the process of developing a simple BCI game based on tic-tac-toe using this framework, in the Python programming language

Tic-tac-toe is used as an example as it is a simple game which everyone knows and understands, but has sufficient complexity to require implementing all the components needed for a working utopia-based BCI, namely: Presentation, Selection and Output.

The overview of the operation of a BCI and the roles of each of these components is covered in the “Utopia : Guide for Implementation of new Presentation and Output Components”, but briefly:



To briefly describe this schematic, the aim of a BCI in general is to control an output device (in this case a robot arm) with your thoughts. In this specific case we control the robot arm by selecting actions perform as flickering virtual buttons on a tablet screen. (See [Mindaffect LABS](#) for a video of the system in action) :

1. **Presentation:** displays a set of options to the user, such as which square to pick in a tic-tac-toe game, or whether to pick something up with a robot arm.
2. Each of the options displayed to the user then flickers with a given unique flicker sequence (or stimulus-sequence) with different objects getting bright and dark at given times.
3. The user looks at the option they want to select to select that option.
4. The users brain response to the presented stimulus is measured by EEG - due to the users focus on their target object, this EEG signal contains information on the flicker-sequence of the target object.
5. The recognition system uses the measured EEG and the known stimulus sequence to generate predictions for the probability of the different options being the users target option.
6. **Selection** takes the predictions from the recogniser and any prior knowledge of the application domain (such as a language model when spelling words) to decide when an option is sufficiently confidently predicted to be selected and output generated.
7. Finally, **Output** generates the desired output for the selected option.

Python is used for the tutorial as the language is well known by most developers and relatively readable and concise allowing us to focus on the core steps needed rather than the language details.

## Initial setup

The first step in development is to set up your development environment. As we will be using python + utopia we will need a copy of the utopia-SDK and a python install.

### Python

For the python install you will need:

- Python 3.X -- (though 2.7 should work)

### Utopia SDK

You can download the Utopia-SDK from here XXXXX. Or request it directly from the developers [jason@mindaffect.nl](mailto:jason@mindaffect.nl).

When you have downloaded the SDK you should extract it to some directory on your development desktop.

In addition the Utopia-SDK requires the following additional components:

- JavaVM - at least java-8. openJVM or oracle-JVM are fine

### Start the FakePresentation (FakeSelection)

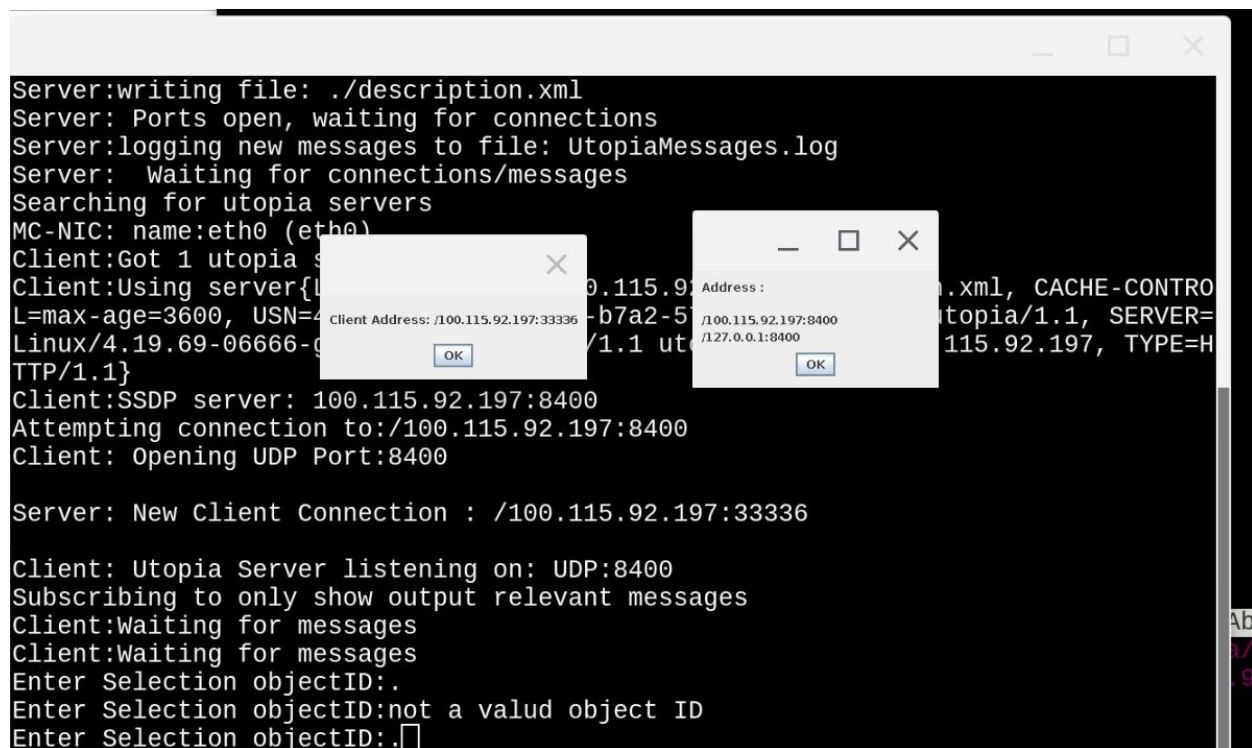
In this tutorial, we focus solely on the output part of the BCI, ignoring the presentation side. However, the event-driven nature of the BCI means that selections cannot happen unless

stimuli are also presented from which the recogniser can generate predictions, and eventually selections.

During this tutorial we will not be using a 'real' presentation system and BCI with EEG attached etc., as that makes debugging difficult. Instead we will use a so-called 'fake-presentation' which simulates the operation of the presentation+selection+EEG+amplifier+recogniser+utopia-hub components. Specifically, as output is only generated when the system is in prediction mode we use a specialized version of this which simply generates fake selections. This component is written in JAVA you can run it by:

1. Running the startup script: `fakeselection.sh` (linux) `fakeselection.bat` (windows)

If successful, this will open a terminal window similar to that shown below, with some initial text about the location of the Utopia-HUB, and window listing the IP addresses you can use to connect to the utopia-hub, and then a prompt waiting for you to enter objectIDs for which selection messages will be generated, and hence output initiated by the script developed here.



```
Server:writing file: ./description.xml
Server: Ports open, waiting for connections
Server:logging new messages to file: UtopiaMessages.log
Server: Waiting for connections/messages
Searching for utopia servers
MC-NIC: name:eth0 (eth0)
Client:Got 1 utopia s
Client:Using server{
L=max-age=3600, USN=4
Linux/4.19.69-06666-g
TTP/1.1}
Client:SSDP server: 100.115.92.197:8400
Attempting connection to:/100.115.92.197:8400
Client: Opening UDP Port:8400

Server: New Client Connection : /100.115.92.197:33336

Client: Utopia Server listening on: UDP:8400
Subscribing to only show output relevant messages
Client:Waiting for messages
Client:Waiting for messages
Enter Selection objectID:.
Enter Selection objectID:not a valid object ID
Enter Selection objectID:.
```

## Main Steps in developing a BCI controlled Device

The main steps in developing a BCI controlled device using utopia are:

1. Connect to the Utopia-Hub
2. Add code to listen for **SELECTION** messages,
3. Filter the selection messages for only those objectIDs for which we should trigger output  
For example, if objectIDs 64,65,66,67 are used to move the robot forward,left,right and backward, filter so only these selections cause output.

4. Add the code to initiate the correct output when the correct selection is detected, i.e. 64->move-forward, 65->turn-right, etc..

## Connect to the Utopia-HUB

As part of the python SDK we provide a utopia-client in the file python/utopiaclient.py. This file contains the code necessary to connect to the utopia hub and send and receive all the utopia message types. Further the Utopia-HUB supports auto-discovery using SSDP. Thus, using this framework you can connect to the utopia-HUB using:

```
from utopiaclient import *
client = UtopiaClient()
client.autoconnect()
```

If you have the fakeslection running this will connect to it and generate some information on the connection like this.

```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from utopiaclient import *
>>> client=UtopiaClient()
>>> client.autoconnect()
Trying to auto-discover the utopia-hub server
Sending query message:
b'M-SEARCH * HTTP/1.1\r\nHOST: 239.255.255.250:1900\r\nMAN:
"ssdp:discover"\r\nST: utopia/1.1\r\nMX: 5\r\n\r\n'
Waiting responses
Got response from : ('100.115.92.197', 60820)
HTTP/1.1 200 OK
USN: 875ad245-f086-4346-b29b-4ed0f4da2567
ST: utopia/1.1
LOCATION: http://100.115.92.197/description.xml
CACHE-CONTROL: max-age=3600
SERVER: Linux/4.19.69-06666-g6c4f8cbba24e UPnP/1.1 utopia/1.1
EXT:

Response matches server type: utopia/1.1
Got location: 100.115.92.197
Loc added to response list: 100.115.92.197
```

```
Discovery returned 1 utopia-hub servers
Discovered utopia-hub on 100.115.92.197 ...
Connected!
>>>
```

Once you are connected to the Utopia-HUB you can send messages using:

```
client.sendMessage(Subscribe(client.getTimeStamp(),"S"))
```

This will send a SUBSCRIBE message, which in this case subscribes this client to only receive SELECTION ("S") messages, which are the only message type needed for an output-only client.

## Add code to listen for **SELECTION** messages,

To receive all messages use:

```
newmsgs=client.getNewMessages(timeout_ms=1000);
```

This will get return any new messages in the message queue, or wait for a maximum of 1000ms for a new message to arrive.

Running this code and printing the messages you should see something like:

```
>>> newmsgs=client.getNewMessages(timeout_ms=1000)
>>> [str(m) for m in newmsgs]
['H(72) HEARTBEAT 58630', 'H(72) HEARTBEAT 60642', 'H(72) HEARTBEAT 62654',
'H(72) HEARTBEAT 64672', 'H(72) HEARTBEAT 66687', 'H(72) HEARTBEAT 68711',
'H(72) HEARTBEAT 70722', 'H(72) HEARTBEAT 72731', 'H(72) HEARTBEAT 74737',
'H(72) HEARTBEAT 76749', 'H(72) HEARTBEAT 78762', 'H(72) HEARTBEAT 80775',
'H(72) HEARTBEAT 82786', 'H(72) HEARTBEAT 84797', 'H(72) HEARTBEAT 86807',
'H(72) HEARTBEAT 88822', 'H(72) HEARTBEAT 90830', 'H(72) HEARTBEAT 92838',
'H(72) HEARTBEAT 94852', 'H(72) HEARTBEAT 96860', 'H(72) HEARTBEAT 98872',
'H(72) HEARTBEAT 100882', 'H(72) HEARTBEAT 102895', 'H(72) HEARTBEAT
104907', 'H(72) HEARTBEAT 106918', 'H(72) HEARTBEAT 108931', 'H(72)
HEARTBEAT 110947']
>>>
```

After inserting a selection event using the fakeselection window (press 8 then enter), we have:

```
>>> newmsgs=client.getNewMessages()
>>> [str(m) for m in newmsgs]
['H(72) HEARTBEAT 199386', 'H(72) HEARTBEAT 201396', 'H(72) HEARTBEAT
203403', 'S(83) SELECTION 203448 id:8']
>>>
```

Here you see we have received a few HEARTBEAT messages, and one SELECTION message:

- The HEARTBEAT messages are used to track if the utopia-HUB is still 'alive' and are sent to all clients even if they have not subscribed to them.
- The SELECTION messages are what we are more interested in. In the display you can see they have 2 fields:
  - timeStamp : this is the time at which the selection happened, measured in milliseconds<sup>1</sup>.
  - objectID : this is the unique object identifier of the object which the system has predicted the user has selected.

To make this work we need to put this waiting for new messages into a loop and then call the function to process the messages:

```
while True:
    newmsgs=client.getNewMessages()
    doAction(newmsgs)
```

## Filter the selection messages

When designing the presentation system, we have agreed that the objectIDs 64,65,66, and 67 will be used exclusively to control the robot. As we are making the robot controller this means we only care about these objectIDs. Thus we filter the selection messages to only respond to these messages.

We could do this in many ways, however as each selection should result in execution of a particular piece of code it is more elegant to use a dictionary containing the set of valid objectIDs and the functions to execute:

```
def doAction(msgs):
    for msg in msgs:
        # skip non-selection messages
        if not msg.msgID==Selection.MSGID :
            continue
        try:
            # get the function to execute for selections we are responsible for
            selectionAction = objectID2Actions[msg.objID]
            # run the action function
            selectionAction(msg.objID)
        except KeyError:
            # This is fine, just means it's a selection we are not responsible for
            Pass
```

---

<sup>1</sup> from some arbitrary 0 point (commonly the session start)

## Add the code to initiate the correct output

The final step is the bit that is unique to this output device. This requires us to do 2 things, firstly to actually write the code which does the output, and secondly fill in the objectID2Actions dictionary to connect these actions to their triggering object identifier.

As we will be controlling a lego-boost robot in the directions forward/left/right and reverse, we first define these action functions:

```
def forward(objID):  
    print('move forward')  
def backward(objID):  
    print('move backward')  
def left(objID):  
    print('move left')  
def right(objID):  
    print('move right')
```

We then connect them to their trigger object identifiers:

```
objectID2Actions = { 64:forward, 65:backward, 66:left, 67:right }
```

## Testing the output component (fakeSelection)

We can test if this is working correctly, by manually inserting the appropriate selection messages from the fakeSelection window. This gives:

Running this, with the relevant selection insertions gives:

In the fakeSelection Window:

```
Server: New Client Connection : /100.115.92.197:34926  
  
Enter Selection objectID:.64  
Enter Selection objectID:.65  
Enter Selection objectID:.66  
Enter Selection objectID:.67  
Enter Selection objectID:.64  
Enter Selection objectID:.Server:  
Error in isReadable : couldnt read any data!  
Server:Read error:Server:Client closed connection!  
Enter Selection objectID:.
```

With the output in the output client window:

```
Response matches server type: utopia/1.1
Got location: 100.115.92.197
Loc added to response list: 100.115.92.197
Discovery returned 1 utopia-hub servers
Discovered utopia-hub on 100.115.92.197 ...
Connected!
move forward
move backward
move left
move right
move forward
```

## Testing the output component, BCI

To test this system with a real BCI, all we need is a real BCI presentation system which has a flicker screen with options for the trigger object IDs (i.e. 64-67).

## Putting it all together, the final code.

Putting all these bits together, including a main loop which constantly waits for messages and generates the final output the implementation is:

```
from utopiaclient import *

def doAction(msgs):
    for msg in msgs:
        # skip non-selection messages
        If not msg.msgID==Selection.msgID :
            continue
        try:
            # get the function to execute for selections we are responsible
            for
                selectionAction = objectID2Action[msg.objID]
                # run the action function
                selectionAction(msg.objID)
        except KeyError:
            # This is fine, just means it's a selection we are not
            responsible for
            Pass
```



```

# the set of actions to perform
def forward(objID):
    print('move forward')
def backward(objID):
    print('move backward')
def left(objID):
    print('move left')
def right(objID):
    print('move right')

# map from objectIDs to the function to execute
objectID2Action = { 64:forward, 65:backward, 66:left, 67:right }

client = UtopiaClient()
client.autoconnect()
client.sendMessage(Subscribe(client.getTimeStamp(),"S"))
while True:
    newmsgs=client.getNewMessages()
    doAction(newmsgs)

```

## Alternative : utopia2output.py

The tutorial so-far talked through the steps required to make an output module 'from scratch'. However, as writing different output modules is such a common thing, we provide a Utopia2Output class which manages the message passing, allowing you to focus on only the output. The simplest way to use this class is by providing your own objectID2Action dictionary:

```

from utopia2output import *
# define our action functions
def forward(objID):
    print('move forward')
def backward(objID):
    print('move backward')
def left(objID):
    print('move left')
def right(objID):
    print('move right')

#We then connect them to their trigger object identifiers
objectID2Action = { 84:forward, 85:backward, 86:left, 87:right }

```

```
# create the output object
utopia2output=Utopia2Output(outputPressThreshold=None)
# replace the default action dictionary with ours. N.B. be careful
utopia2output.objectID2Action=objectID2Action
# run the output module
utopia2output.run()
```