# Tutorial: How to make a simple BCI selection grid

Ver 1.0 29/10/19

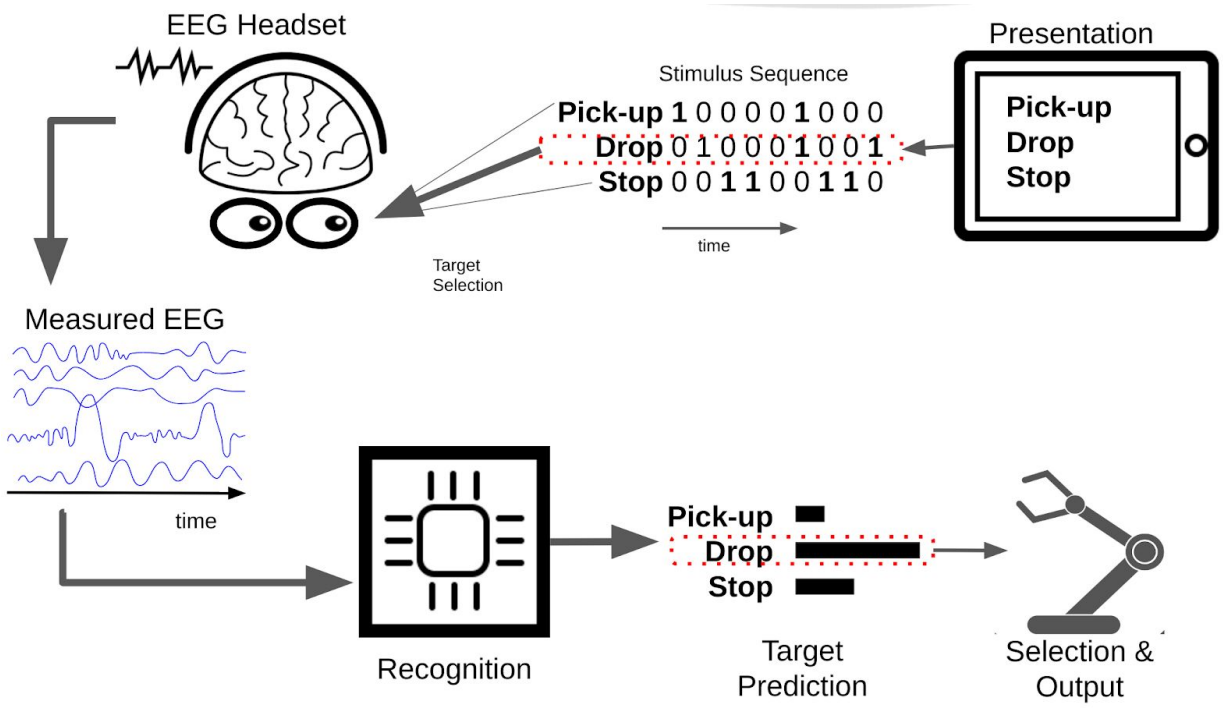


# Introduction

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)
- Pyglet -- we will use this for presenting the game to the user and flickering the objects.

## Utopia SDK

You can download the Utopia-SDK from here XXXXX. Or request it directly from the developers at 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:

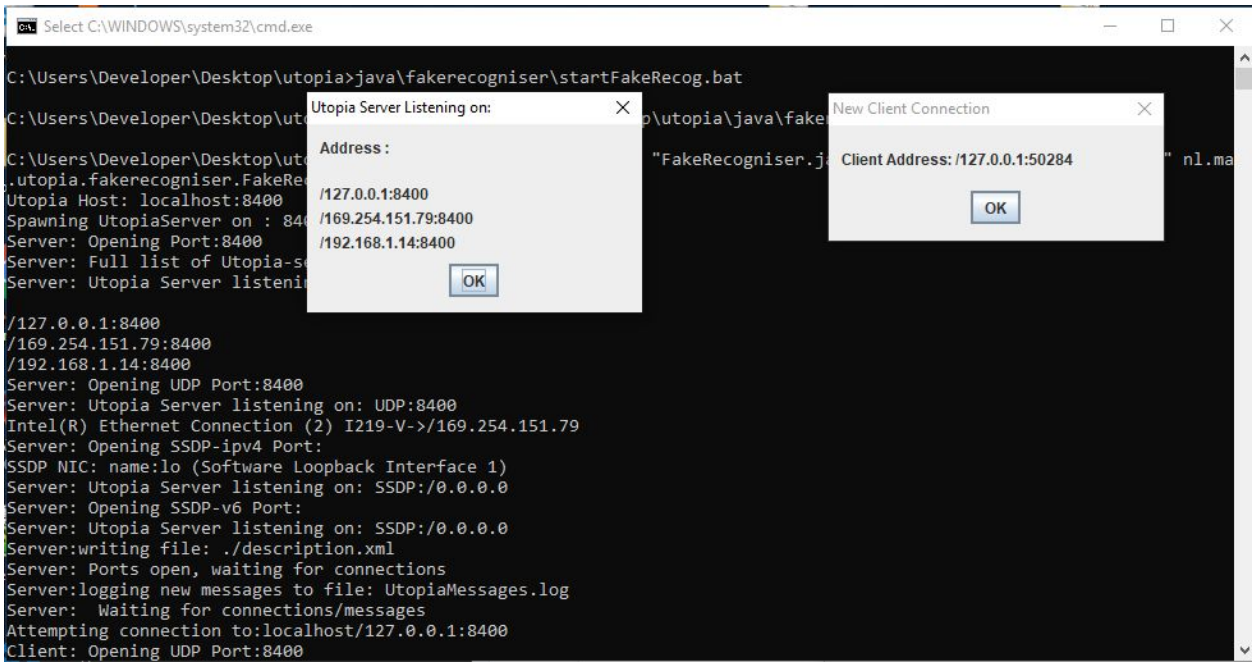
- JavaVM - at least java-8. openJVM or oracle-JVM are fine
- Python - at least version 3.0. Get from: https://www.python.org/
- pyglet - python graphics library for the stimulus presentation.
  (Install with: `python3 -m pip install pyglet`)

## Start the FakeRecogniser

During this tutorial we will not be using a 'real' BCI with EEG attached etc., as that makes debugging difficult. Instead we will use a so-called 'fake-recogniser' which simulates the operation of the EEG-acquisition+Utopia-HUB+Recogniser components. This component is written in JAVA you can run it either by:

1. Running the startup script: `fakerecogniser.sh` (linux) `fakerecogniser.bat` (windows)
2. Directly running the .jar file found in : java/fakerecogniser/build/jar/fakerecogniser.jar

If successful, this will open a terminal window with lots of text streaming up it, and a window listing the IP addresses you can use to connect to the utopia-hub.
If successful, this will open a terminal window similar to that shown below, with some initial text about the location of the UtopiaHUB, and window listing the IP addresses you can use to connect to the UtopiaHUB, and then a stream of debugging information about the connection and the fakerecogniser state.




To test if this is working correctly you can now directly run the tic-tac-toe game this tutorial is going to teach you by running:

- `tictactoe.bat` (window) or `tictactoe.sh` (linux)

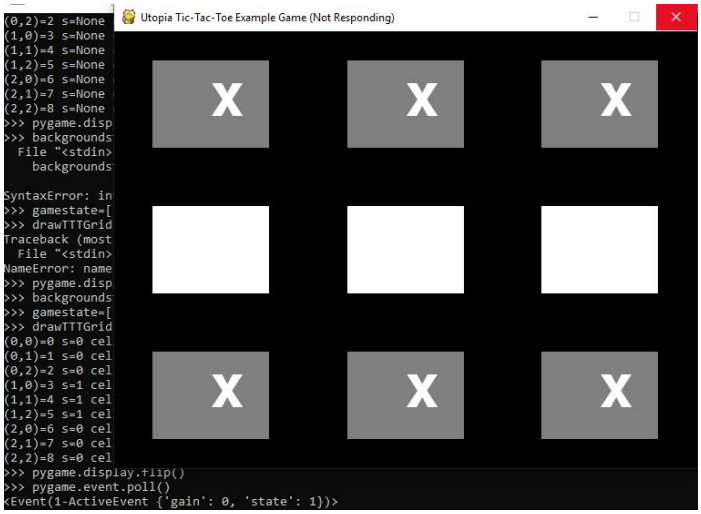
Note: The final code from this tutorial is available in:
`python/messagelib/lettermatrix.py`

This should run through the main stages for using a BCI. Specifically:

1. **Calibration**: During this phase the user is cued with a green square telling them what grid cell to concentrate on. Then this flashes in sequence to get some *training-data* from which the system will learn to recognise this users unique response to the stimulus. This should look something like this:


2. **Prediction**: During this phase the user can select where they want to move by looking at the cell they want to select. What you should see is random X's being selected by the

*simulated* user, followed by O's selected by the computer. This should look something like this:




NOTE: At this point we are **NOT** doing a real Brain Computer Interface, as there is no actual brain attached, so results will be RANDOM!!

# Main Steps in developing the selection grid

The main steps in developing a BCI using utopia are:

1. Develope the on-screen display
2. import to `fakerecogniser.py` and override the draw function
3. Connect to the Utopia-Hub, run calibration and prediction

NOTE: to run this tutorial as is, without any code adaptations, please save the python file in `python/messagelib/`

As an example of the use of this framework, we will develop a simple letter-matrix selection system.

NOTE: you can find a full working example of this code in `python/messagelib/matrixspeller.py`

## Develop the stimulus display code.

This is the most complex part of the development, as we must make the display and the code to draw the display according to the current flicker state. We do this in 2 parts, a) initialize the display, b) draw the display according to the flicker state.

The code to initialize the display is given here.

```
#--------------------------------------------------------------------
```

```
# Initialization : display, utopia-connection
def init(symbols,bgFraction=.2):
    '''Intialize the stimulus display with the grid of strings in the
    shape given by symbols.
    Store the grid object in the fakepresentation.objects list so can
    use directly with the fakepresentation BCI presentation wrapper.'''
    global window, batch, objects, labels

    window = pyglet.window.Window(width=1024,height=768,vsync=True)

    winw,winh=window.get_size()

    # create set of sprites and add to render batch
    batch = pyglet.graphics.Batch()
    background = pyglet.graphics.OrderedGroup(0)
    foreground = pyglet.graphics.OrderedGroup(1)

    # get size of the matrix
    gridwidth  = len(symbols)
    gridheight = len(symbols[0])
    ngrid      = gridwidth * gridheight

    # add a background sprite with the right color
    objects=[None]*ngrid
    labels=[None]*ngrid
    w=winw/gridwidth # cell-width
    bgoffsetx = w*bgFraction
    h=winh/gridheight # cell-height
    bgoffsety = h*bgFraction
    for i in range(gridheight): # rows
        y = i/gridheight*winh # top-edge cell
        for j in range(gridwidth): # cols
            idx = i*gridwidth+j
            x = j/gridwidth*winw # left-edge cell
            # create a 1x1 white image for this grid cell
            img =
pyglet.image.SolidColorImagePattern(color=(255,255,255,255)).create_image(1
,1)
            # convert to a sprite (for fast re-draw) and store in objects
list
            # and add to the drawing batch (as background)
```

```
objects[idx]=pyglet.sprite.Sprite(img,x=x+bgoffsetx,y=y+bgoffsety,batch=bat
ch,group=background)
            # re-scale (on GPU) to the size of this grid cell

objects[idx].update(scale_x=int(w-bgoffsetx*2),scale_y=int(h-bgoffsety*2))
            # add the foreground label for this cell, and add to drawing
batch

labels[idx]=pyglet.text.Label(symbols[i][j],font_size=32,x=x+w/2,y=y+h/2,co
lor=(255,255,255,255),anchor_x='center',anchor_y='center',batch=batch,group
=foreground)
    return objects
```

Basically, this just creates a grid of images with associated text labels and returns them in the objects list. It also initializes important global variables.

The code to draw the display according to the flicker state is given here.

- **Note**: When used with fakepresentation, this function will be called once per video-frame with the updated stimulus-state to allow you to re-draw the display with your flickering objects with the given flicker state. If you want a dynamic display or more complex display logic then you should put it into this function (or functions you call from it.).

```
#--------------------------------------------------------------------
# override functions to make the screen refresh
def draw(stimulusstate=None):
    """draw the letter-grid with given stimulus state for each object.
       Note: To maximise timing accuracy and the stimulus update rate, we
             'flip' the screen as the last line, which implicitly waits
             for the screen vertical-blanking from the hardware.  Hence
             maximising the timing accuracy."""
    # draw the white background onto the surface
    window.clear() #getColor('idle'))
    # update the state
    for idx in range(len(fp.objects)):
        # get the background state of this cell
        bs = stimulusstate[idx] if stimulusstate else None
        # color depends on the requested stimulus state
        if bs==0 :    # off
            fp.objects[idx].color=(0,0,0)
        elif bs==1 :  # flash
            fp.objects[idx].color=(255,255,255)
        elif bs==2 :  # cue
```

```
            fp.objects[idx].color=(0,255,0)
        elif bs==3 :  # feedback
            fp.objects[idx].color=(0,0,255)
    # do the draw
    batch.draw()
    # wait for the screen flip
    event = window.dispatch_events()
    pyglet.clock.tick(poll=True)
    window.flip()
```

Basically this just loops over the display objects and sets their color according to the flicker state and then as the last step draws the updated display and waits for the display 'flip', i.e. for the screen to refresh.

## Import fake presentation and connect your draw function

```
import fakepresentation as fp
# set the set of objects to flicker in fakepresentation
fp.objects=init([['a','b','c']['d','e','f']])
# set the stimulus re-draw function in fakepresentation
fp.draw=draw
```

## Connect to the utopia-hub, load the stimulus sequence, calibrate and predict.

The final step is to tie the display code into the main experiment, connect to the utopia-hub, run the calibration and then run the prediction. The code to do this is:

```
# (auto)-connect to the utopia-hub
fp.connect2Utopia()
# load the stimulus sequence to use
stimSeq,stimTime_ms=fp.loadStimSequence()
stimTime_ms=fp.setStimRate(stimTime_ms,60)
# run the user calibration phase, that is cued attention,
# with 10-trials of 4 seconds
fp.runCalibration(stimSeq,stimTime_ms,numTrials=10,trialDuration=4)

# run the prediction phase, that is user chosen objects,
# for 10-trials with feedback and with adaptive trial length (max 10s)
fp.runPrediction(stimSeq,stimTime_ms,trialDuration=10,numTrials=10)
```

```
# Alternatively: run a single target selection, with max 10s trial
# and early trial termination of the error of the predicted target is <10%
selectedObjID = playStimulusSequenceWithSelection(
        stimSeq,stimTime_ms,
        duration_ms=10000, # max 10s trial
        selectionThreshold=.1) # early stop if predicted error <10%
```

# Complete fakepresentation example: raspberry PI GPIO presentation

This example, uses the gpio pins on the raspberry PI instead of the screen. Thus it does not need to do any display stuff which makes the code much shorter and less complex than when using the display above.
Note: you can find this code in `python/messagelib/rpigpio.py`

```
import time
import fakepresentation as fp
from gpiozero import LED

#---------------------------------------------------------------------
# override functions to make do the GPIO commands
def draw(stimulusstate=None,targetidx=None):
    """draw the tic-tac-toe grid, with given game-state (x/o's) and
background-state"""
    #print("Background state"+str(backgroundstate))
    # BODGE: sleep to limit the stimulus update rate
    time.sleep(1/framerate)
    # update the state of each LED to match the stimulusstate
    for idx in range(len(fp.objects)):
        # get the background state of this cell
        bs = stimulusstate[idx] if stimulusstate else None
        if not bs is None and bs>0 :
            fp.objects[idx].on()
        else :
            fp.objects[idx].off()

if __name__=="__main__":
    # set the flickering objects in fake-presentation to be the GPIO pins
    fp.objects=[LED(i) for i in [2,3,4]]
    # override the draw functions from fake presentation
    framerate=65
```

```
    fp.draw=draw # override draw from fake presentation
    fp.showInstruct(["Searching for the utopia-HUB","","Please
wait"],timeout_ms=0)
    # connect to the utopia-hub
    fp.connect2Utopia()

    # load the stimulus sequence
    stimSeq,stimTime_ms=fp.loadStimSequence()
    stimTime_ms=fp.setStimRate(stimTime_ms,60)

    # do the brain-response calibration (i.e. with user cueing) for
10trials

fp.runCalibration(stimSeq,stimTime_ms,numTrials=30,trialDuration=10,cueDura
tion=4)

    # do the prediciton (with feedback) for 10 trials
    fp.runPrediction(stimSeq,stimTime_ms,trialDuration=10,numTrials=10)
```

# Summary

Making a visual-evoked potential BCI game with the Utopia framework is relatively straight forward. The main steps you need to take are:

1. add the flicker-sequence,
2. add the STIMULUSEVENT messages,
3. add the calibration phase (with target information to the STIMULUSEVENT messages)
4. add the PREDICTEDTARGETPROB listening, selection, and early trial termination,
5. add finally adding mode-changing logic

# Further enhancements

If this tutorial has inspired you to go further in building a BCI game, here are a few ideas for enhancements:

1. Make it work with another game. Tic-Tac-Toe is a simple game with very simple graphics which makes it good for a tutorial as we can focus more on the Utopia integration aspects. However, it's "a bit boring". However, now you know the main steps you need to take, you can think about how to make a more interesting game. How about Chess, or Space-Invaders, or something in virtual-reality, or making a 2-player game with one Brain and one finger player?

2. Make it work to control other devices. Here the output was simply to play a move on the on-screen game board. However, you can use the selections to make any arbitary output happen. How about using the BCI to control a lego robot (Mindstorms? Boost (we recommend using pylgbst)).
3. Make a nicer framework. The code developed here is mainly for teaching purposes. Thus it exposes quite a lot to the developer, and makes some *dubious* design choices for simplicity of explanation. In particular having the message management running on the main drawing thread is a bit dubious, as is having many messages silently discarded if they are not PREDICTEDTARGETPROB messages…. It would be good to re-design these into a proper framework to make game development easier / more robust.