

# Tutorial: How to make a tic-tac-toe BCI game

Ver 1.0

29/10/19

## [Introduction](#)

### [Initial setup](#)

#### [Python](#)

#### [Utopia SDK](#)

#### [Start the FakeRecogniser](#)

## [Main Steps in developing the BCI game](#)

### [Connect to the Utopia-HUB](#)

### [Load the desired stimulus-sequence from file](#)

### [Develop the on-screen display and stim-sequence playback code](#)

#### [Display a 3x3 tic-tac-toe grid](#)

#### [Play a stimulus sequence on the 3x3 grid](#)

### [Add the code to send STIMULUSEVENT information to the Utopia-Hub](#)

### [Add Calibration mode specific logic](#)

#### [Tell the recogniser we entering/leaving calibration mode](#)

#### [Add Cues to the stimulus display to tell the user where to look.](#)

#### [Add information to tell the recogniser which object is the target](#)

#### [Wrap it all up in a loop over targets of fixed duration](#)

#### [Testing Calibration Mode](#)

## [Add Prediction mode specific logic](#)

### [Listening to the PREDICTEDTARGETPROB messages](#)

### [Select a target when the current prediction is sufficiently confident](#)

### [Run the flicker sequence and terminate early if a selection is made](#)

### [Output the selection to the user and make the AI moves.](#)

## [Testing the tic-tac-toe BCI game](#)

## [Summary](#)

## [An easy to use BCI Presentation framework: fakepresentation](#)

### [Develop the stimulus display code.](#)

### [Connect the display to the fakepresentation](#)

### [Connect to the utopia-hub, load the stimulus sequence, calibrate and predict.](#)

## [Complete fakepresentation example: raspberry PI GPIO presentation.](#)

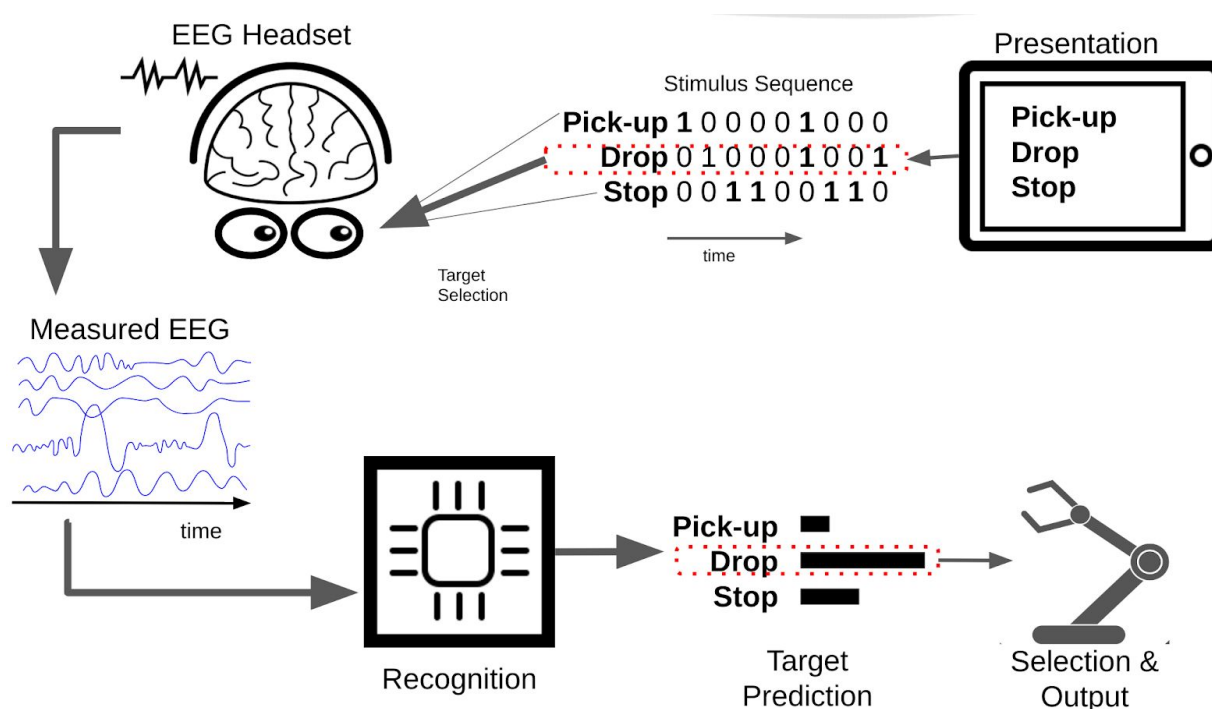
[Further enhancements](#)

## 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](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
- 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`)

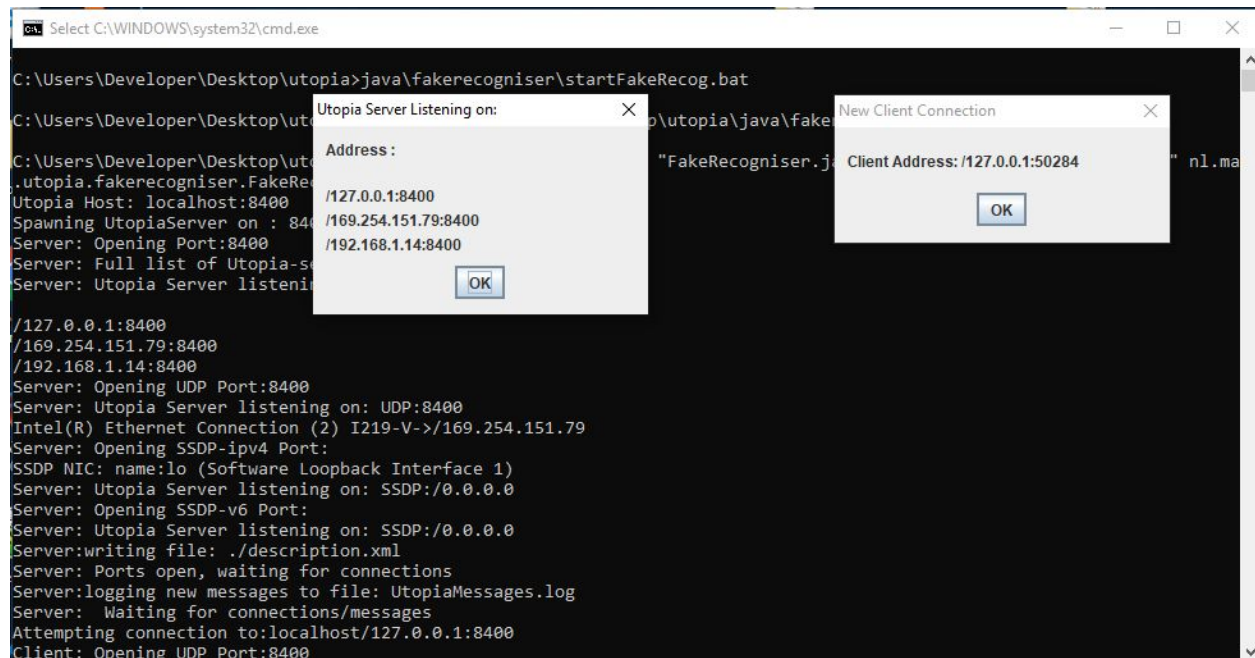
## Start the FakeRecogniser

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:

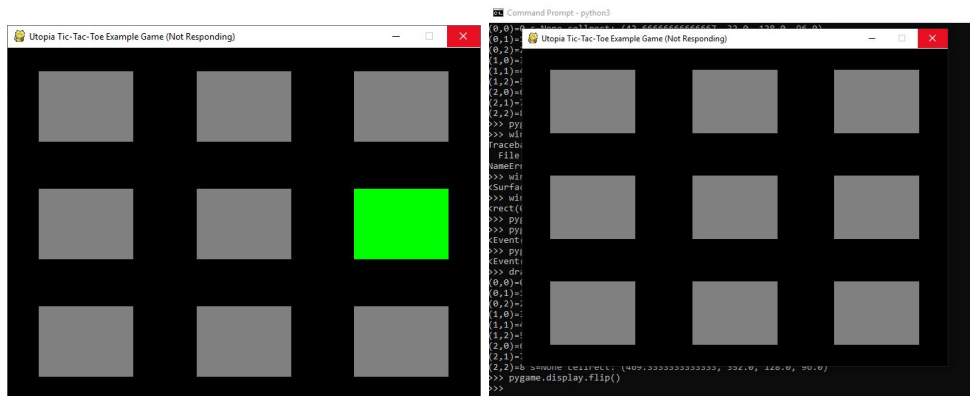
- `tictactoe.bat` (window) or `tictactoe.sh` (linux)

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

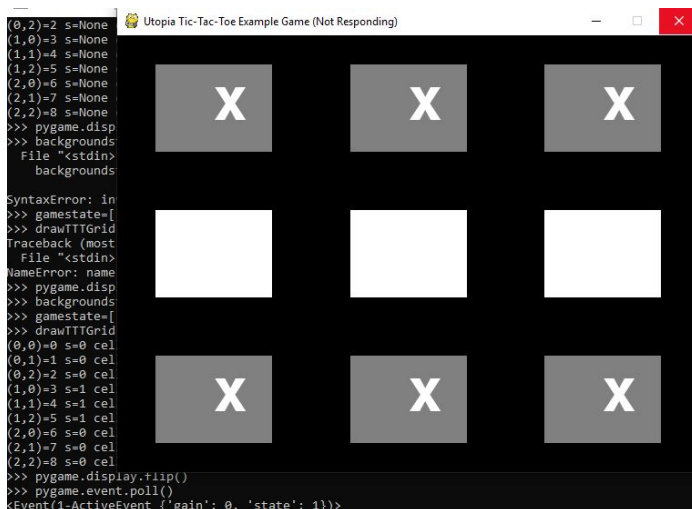
This should run through the main stages for using a BCI. Specifically:

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 BCI game

The main steps in developing a BCI using utopia are:

1. Connect to the Utopia-Hub
2. Load the desired stimulus-sequence from file
3. Develop the on-screen display and stimulus-sequence display code
4. Add the code to send STIMULUSEVENT information to the Utopia-Hub

5. Add Calibration mode specific logic -- specifically to cue the user to look at a specific target when in calibration mode, and add the cue information to the STIMULUSEVENT messages.
6. Add Prediction mode specific logic - specifically, code to listen for PREDICTEDTARGETPROB messages and select a target when the prediction is sufficiently confident and output that selection to the user and update the game state.
7. Add the mode-switching logic to switch between calibration mode (where the user is cued where to look) and prediction mode (where the user is able to make their own selections), and send the appropriate MODECHANGE messages.

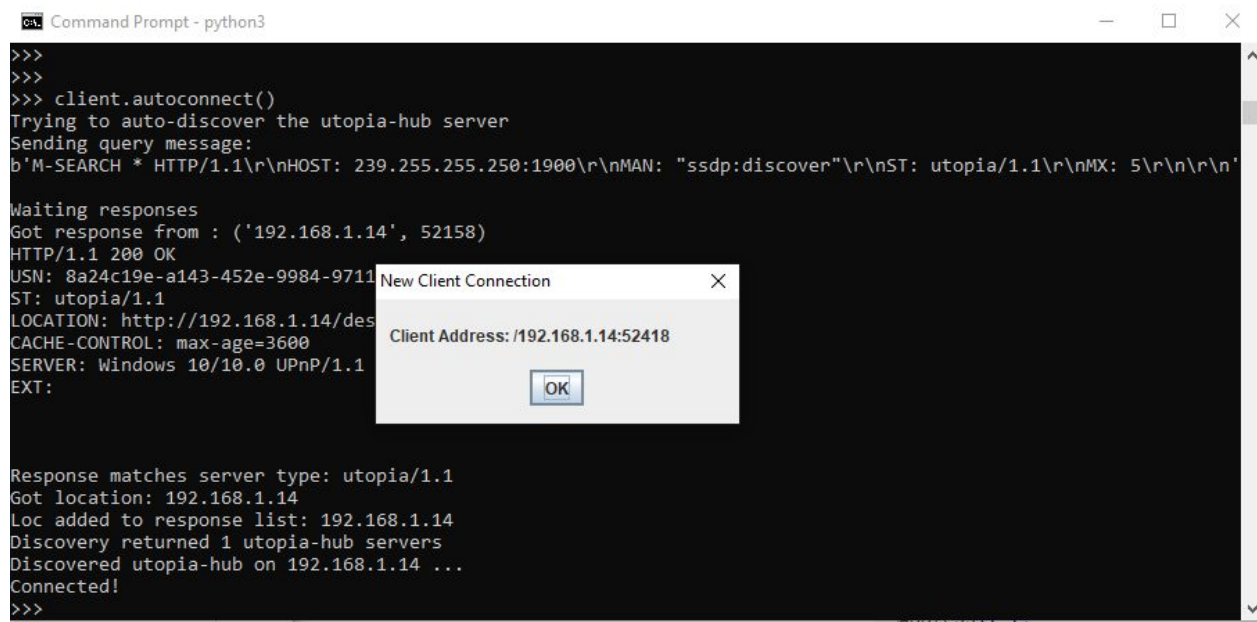
NOTE: to run this tutorial as is, without any code adaptations, please save the python file in `python/messageolib/`

## 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 fakerecogniser running this should give output something like this:



```

C:\> Command Prompt - python3
>>>
>>> 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 : ('192.168.1.14', 52158)
HTTP/1.1 200 OK
USN: 8a24c19e-a143-452e-9984-9711
ST: utopia/1.1
LOCATION: http://192.168.1.14/des
CACHE-CONTROL: max-age=3600
SERVER: Windows 10/10.0 UPnP/1.1
EXT:

Response matches server type: utopia/1.1
Got location: 192.168.1.14
Loc added to response list: 192.168.1.14
Discovery returned 1 utopia-hub servers
Discovered utopia-hub on 192.168.1.14 ...
Connected!
>>>

```

Where as you can see, if successful we finish with a 'Connected!' message.

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

```
client.sendMessage(Subscribe(client.getTimestamp(),"PMSN"))
```

This will send a SUBSCRIBE message, which in this case subscribes this client to only receive PREDICTEDTARGETPROB ("P"), MODECHANGE ("M"), SELECTION ("S") and NEWTARGET ("N") messages.

To receive 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. (Set timeout\_ms=0 to block until any messages are received.). The number of messages depend on the last time you have asked for the new messages. If no new messages are detected within this time, extend the number of ms.

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']
>>>
```

As you can see, we only have Heartbeat messages (nothing else is running) which are provided to track the state of the UtopiaHUB and our connection to it. Each message has;

- a type (H or integer 72 for Heartbeats),
- a time-stamp, which is a integer representing the real-time in milliseconds from some arbitrary starting time.
- Some additional arguments (which vary depending on the message type).

## Load the desired stimulus-sequence from file

For an evoked response BCI it is important that the right 'type' of flicker-sequence be used so that it evokes the expected type of response in the users brain when the look at their desired option. For Utopia by default this sequence should evoke a code-Visual-Evoked-Potential, (or c-VEP). As the design of flicker-sequences can be complex the Utopia-SDK provides a set high



performance flicker-sequences developed by MindAffect along with code to load these sequences from a simple text file format. To load the default noisecode use:

```
from stimseq import StimSeq
noisecode =
StimSeq.fromFile("../resources/codebooks/mgold_65_6532_psk_60hz.txt")
```

You can check what has been loaded by running:

```
>>> noisecode.stimSeq[1:5][1:5]
[[0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0,
 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0,
 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0,
 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0,
 1.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0,
 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0,
 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0,
 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0,
 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0,
 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0,
 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0,
 1.0, 0.0, 1.0]]
>>> noisecode.stimTime_ms[1:5]
[17.0, 33.0, 50.0, 67.0]
>>>
```

The noisecode contains the following fields:

- `stimSeq` - which is a list of lists of size  $[T \times S]$  for  $T$  time points and  $S$  options. The entry `stimSeq[t][s]` gives the stimulus state for the  $s$ 'th option at the  $t$ 'th time point.
- `stimTime_ms` - which is a list of size  $[T \times 1]$  for  $T$  time-points. The entry at `stimTime_ms[t]` gives the time in milliseconds from the start of the sequence at which presentation should change to that of `stimSeq[t]`. (That is at time `stimTime_ms[t]` the presentation state should change to `stimSeq[t]`).

## Develop the on-screen display and stim-sequence playback code

We are making a simple tic-tac-toe game. The display should consist of:



1. Display; should be a simple 3x3 grid (with the # of grid-boundary lines?) with background squares that flicker. (Note: we make the whole letter background flicker rather than just the letter itself so empty cells without an x-or-o can be selected.)
2. For already selected squares, the character X or O depending on which player selected that square.
3. For playing the stimulus sequence; each of the (non-selected) options flickering with a unique stimulus sequence from the loaded stimulus-sequence file.

As accurate timing of the changes in the visual display is **very important** for an evoked response BCI. Let me say that again to be sure to make the point:

## Accurate timing of the **changes** in the visual display is **very important** for a visual evoked response BCI

Thus, we need to be sure to use a display system which provides fast and accurate information on display changes. At MindAffect we have tested a few different pythonic frameworks for visual display and decided on [pyglet](#) as it gives high visual timing performance whilst being both relatively robust cross platform<sup>1</sup> (though dependent on the video hardware you use) and relatively easy to use.

In use pyglet the command `window.flip()` is used to initiate a display update and (importantly for timing accuracy) **waits** for the actual display to change (a so-called vsync) before returning. Thus, from the python we know that immediately after this line has happened the display has been redrawn and can use the time of this happening to accurately control (and measure) when the display changes happened.

### Display a 3x3 tic-tac-toe grid

The detail of implementing a pyglet based display not so important here. So we just provide the code below.

To start we need some basic code to initialize the pyglet display, and store relevant context, e.g. the window in some global variables:

```
import pyglet
def initPyglet(fullscreen=False):
    global window, basicFont, instructLabel, keyhandler
    # set up the window
    if fullscreen :
        # N.B. accurate video timing only guaranteed with fullscreen
        config = pyglet.gl.Config(double_buffer=True)
        window = pyglet.window.Window(vsync=True, config=config)
    else :
        config = pyglet.gl.Config(double_buffer=True)
```

<sup>1</sup> Unfortunately, (at time of writing) pyglet does not work well on the raspberry pi, with poor timing accuracy and lots of image tearing..

```

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

    # setup a key handler
    keyhandler = pyglet.window.key.KeyStateHandler()
    window.push_handlers(keyhandler,on_close)

    # initialize the stimulus screen
    initGrid()
    # initialize the instructions screen

instructLabel=pyglet.text.Label(x=window.width//2,y=window.height//2,anchor
_x='center',anchor_y='center',font_size=24,color=(255,255,255,255),multiline=True,width=int(window.width*.8))
    # do first screen re-draw
    flip()

def on_close():
    '''close the app if the close button is pressed'''
    exit()

```

Basically, this code just starts pyglet, opens a window (with fullscreen=False, set fullscreen=True to get full-screen mode) to display the output and generates two fonts, one big and one small for display of text for the user.

Next we have the code to initialize the grid display. Basically, this just makes a set of 9 images and text-labels, one for each cell of the 3x3 tic-tac-toe grid and positions them in the correct place. It also puts all the 'drawing objects' into a so-called '*batch*' which is used by pyglet to rapidly draw the whole screen.

```

def initGrid(bgFraction=.2):
    global batch, background, foreground, sprites, labels, opto
    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)
    # add a background sprite with the right color

bg=pyglet.sprite.Sprite(pyglet.image.SolidColorImagePattern(color=(128,128,
128,255)).create_image(1,1),x=0,y=0,batch=batch,group=background)
    bg.update(scale_x=winw,scale_y=winh)

```

```

sprites=[None]*9
labels=[None]*9
w=winw/3 # cell-width
bgoffsetx = w*bgFraction
h=winh/3 # cell-height
bgoffsety = h*bgFraction
for i in range(3): # rows
    y = i/3*winh # top-edge cell
    for j in range(3): # cols
        idx = i*3+j # linear index of this cell TODO: check order..
        x = j/3*winw # left-edge cell
        sprites[idx] =
pygame.sprite.Sprite(pygame.image.SolidColorImagePattern(color=(255,255,255
,255)).create_image(1,1),x=x+bgoffsetx,y=y+bgoffsety,batch=batch,group=back
ground)
        sprites[idx].update(scale_x = int(w-bgoffsetx*2),scale_y =
int(h-bgoffsety*2))
        Labels[idx] =
pygame.text.Label('+',font_size=32,x=x+w/2,y=y+h/2,color=(255,255,255,255),
anchor_x='center',anchor_y='center',batch=batch,group=foreground)

opto = pygame.sprite.Sprite(
pygame.image.SolidColorImagePattern(color=(255,255,255,255)).create_image(1
,1),x=winw-winw*.1,y=winh-winh*.1,batch=batch,group=background)
opto.update(scale_x=int(winw*.1),scale_y=int(winh*.1))

```

Next we have the function which actually draws the updated grid at run time. This function takes two main inputs.

- **backgroundstate** is a [9x1] list with each entry indicating the stimulus-state of that cell in the grid, where the state is one of: idle, flash, cue or feedback. Each of these states is then mapped to a particular background color using the `getColor()` function with the colormap defined in the variable `colors`
- **gamestate** is a [9x1] list of strings containing the text to be displayed in each grid cell, i.e. X or O depending on if the user (X) or computer (O) has previously selected that cell.
- Note: as we may wish to draw additional information on top of the grid the **`window.flip()`** is not included in this function but run later in a wrapper function.

```

def
drawTTTGrid(backgroundstate=None,gamestate=None,win=None,targetidx=None,opt
oSensor=True,bgFraction=.2):
    """draw the tic-tac-toe grid, with given game-state (x/o's) and
background-state"""

```

```

#print("Background state"+str(backgroundstate))
# draw the white background onto the surface
if win is None: # use global window if none given
    win=window
win.clear() #getColor('idle'))
opto.visible= not targetidx is None
# update the state
for i in range(3): # rows
    for j in range(3): # cols
        idx = i*3+j # linear index of this cell TODO: check order..
        # get the background state of this cell
        bs = backgroundstate[idx] if backgroundstate else None
        # get the background color
        bg = getColor(bs)
        sprites[idx].color=bg
        # draw the optoSensor signal
        if optoSensor and idx==targetidx :
            opto.color=bg
        # draw the foreground text
        if gamestate and gamestate[idx] is not None :
            if not gamestate[idx]==labels[idx].text :
                labels[idx].text=gamestate[idx]
# do the draw
batch.draw()

```

To use drawTTGrid we need a utility function which maps from the description of the background state, as either 0 or 1 or 'cue' or 'feedback' etc., into a color for that cell to be displayed. This is provided by the function getColor:

```

# map from stimulus states to display colors
stimColors={'off':(0,0,0), 'flash':(255,255,255), 'cue':(0,255,0),
'feedback':(0,0,255), 'idle':(128,128,128)}
# map from stimulus state numbers to stimulus states
stimState={0:'off',1:'flash',2:'cue',3:'feedback',4:'idle'}
def getColor(bs):
    """map from background stimulus state to color"""
    if type(bs) is not str : # map to state string
        if bs is None:
            bs='off'
        else:
            bs=stimState[int(bs)]

```

```
col=stimColors[bs]
return col
```

None of the code so-far will actually draw anything to the display. We use the general utility function `flip`, which causes the display to update and (importantly) as mentioned early **waits for the screen to actually redraw**. This utility function then calls another general utility function `pump_events` which causes the pygamelet system to do it's general event processing so the general GUI components are updated, e.g. for the window close button.

```
def flip(waitBlanking=False, getEvents=True):
    '''block current thread until the back-buffer is unlocked for writing
    again,
    generally this means until just after the screen blanking, i.e.
    vsync'''
    window.flip()
    pump_events()

def pump_events():
    window.switch_to()
    event = window.dispatch_events()
    window.dispatch_event('on_draw')
    pygamelet.clock.tick(poll=True)
```

Further we have general a general utility function to show textual instructions to the user:

```
def
showInstruct(text,waitKey=False,timeout_ms=0,win=None,clearScreen=True):
    '''Show a block of text to the user for a given duration on a blank
    screen'''
    if win is None: # use global window if none given
        win=window
    if clearScreen:
        win.clear()
    # tell the user that calibration has finished
    if type(text) is list:
        text = "\n".join(text)
    if not text == instructLabel.text : # only if changed..
        instructLabel.begin_update()
        instructLabel.text=text
        instructLabel.end_update()
    instructLabel.draw()
```

```

# refresh the display
flip()
if waitKey:
    print('Wait for keypress of %f'%(timeout_ms/1000))
    waitForKey(timeout_ms)
elif timeout_ms :
    print('Screen sleep %f'%(timeout_ms/1000))
    waitFor(timeout_ms)

```

Finally, it's useful to have a function to wait the system progress either for a fixed time or for user input. This is provided by the `waitForKey` and `waitFor` functions.

```

def waitForKey(timeout_ms=10000):
    """ Wait for the user to press a key and return the pressed key."""
    print("waitForKey")
    keyhandler.clear()
    t0=getTimeStamp()
    while getTimeStamp(t0) < timeout_ms :
        #print('.',end='')
        if keyhandler.keys() :
            #for k in keyhandler.keys() : print(k,keyhandler[k])
            keys=keyhandler.keys()
            keyhandler.clear()
            return keys
        pump_events()
        time.sleep(.005)#waitFor(100)
    return

def waitFor(timeout_ms):
    """wait for the given time, running the display loop"""
    t0=getTimeStamp()
    while getTimeStamp(t0)<timeout_ms:
        pump_events()

```

To get a relative wall timestamp in milliseconds, you would need the function `getTimeStamp`

```

def getTimeStamp(t0=0):
    '''get a (relative) wall-time stamp *in milliseconds*'''
    if client:
        return client.getTimeStamp()-t0
    return time.perf_counter()*1000-t0

```

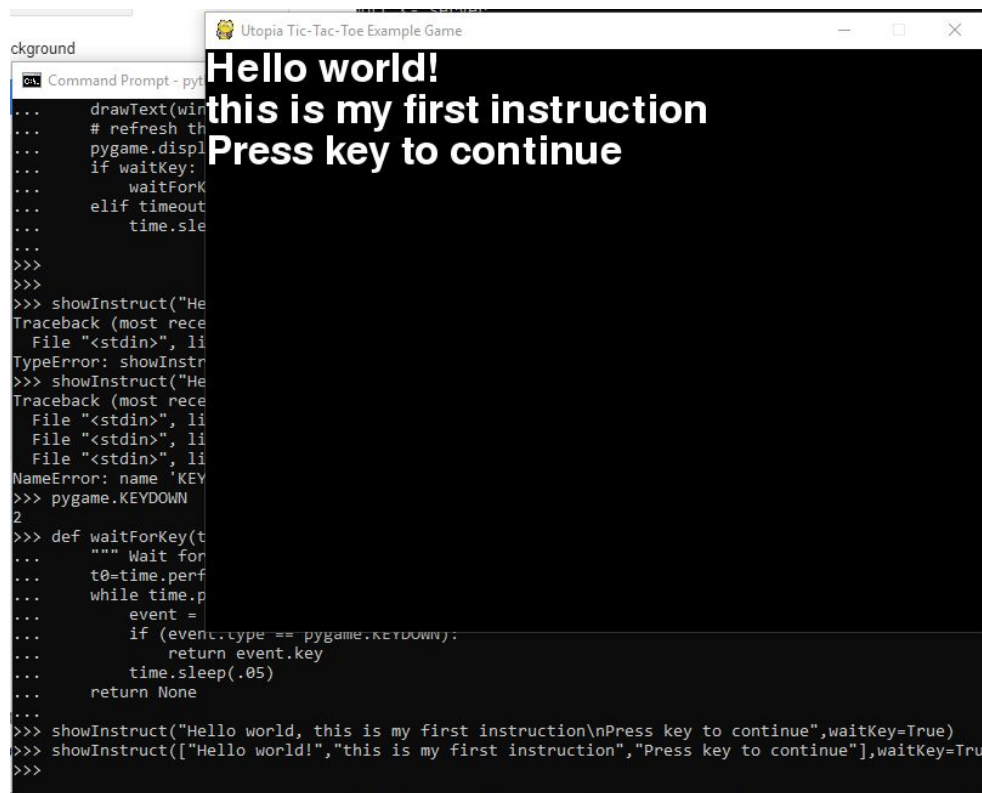
After defining these functions. You can run:

```
initPyglet()
```

Then you can show some instructions with:

```
showInstruct(["Hello world!","this is my first instruction","Press key to  
continue"],waitKey=True)
```

Which should show a window like below and wait for you to press a key.



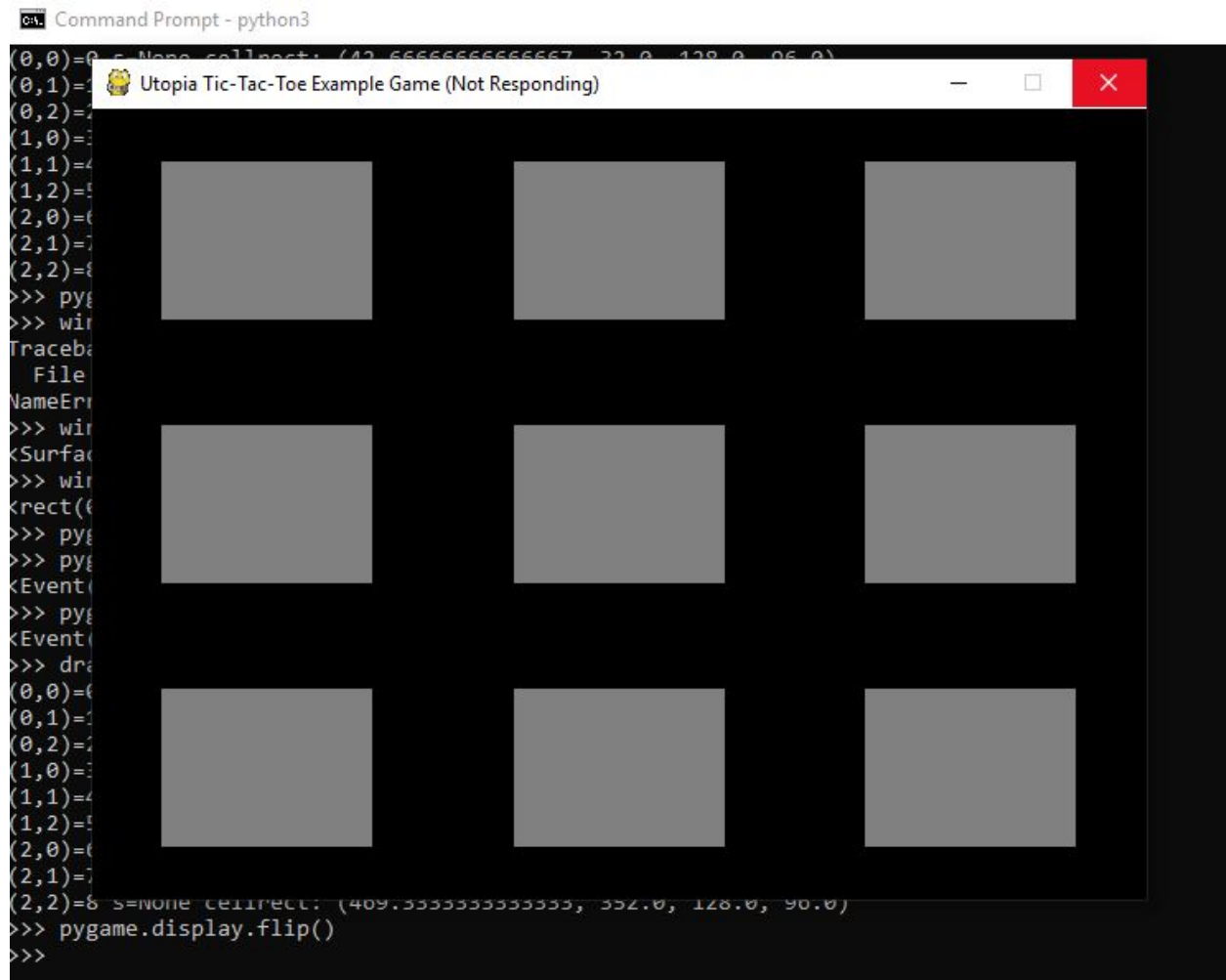
Or show the tic-tac-toe grid with:

```
drawTTTGrid()  
flip()
```

Note the need to explicitly call `flip()` in this case to update the screen. You may also need to call `pump_events()` to ensure the window responds correctly to OS events.

This should show a window like this:

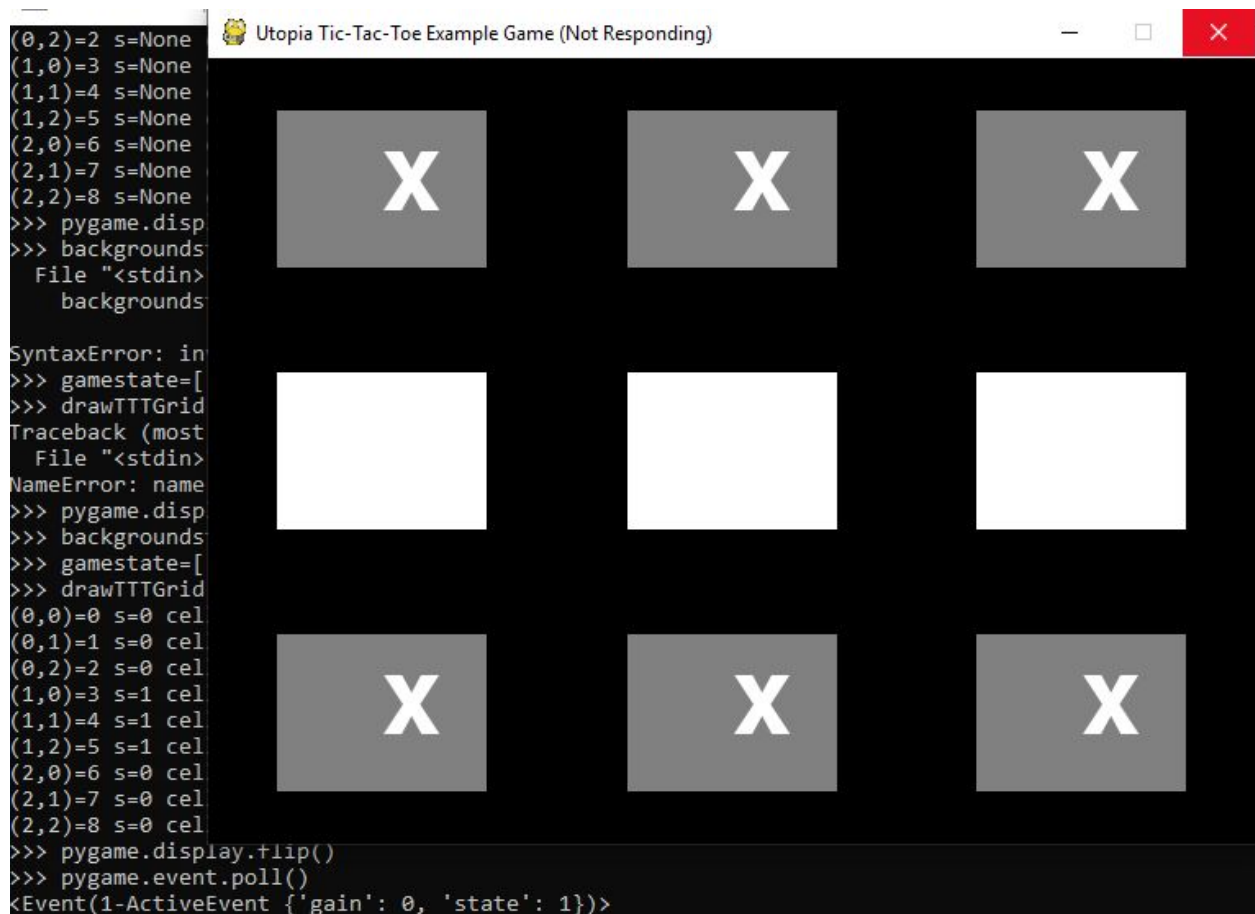




Or including some backgroundstate to highlight the middle row and gamestate information to show that X is about to win:

```
backgroundstate=[0,0,0,1,1,1,0,0,0]
gamestate=['x','x','x','x','o','x','x','x','x']
drawTTTGrid(backgroundstate,gamestate)
flip()
```

Which should look like this:



```

(0,2)=2 s=None
(1,0)=3 s=None
(1,1)=4 s=None
(1,2)=5 s=None
(2,0)=6 s=None
(2,1)=7 s=None
(2,2)=8 s=None
>>> pygame.display
>>> backgrounds
File "<stdin>"
backgrounds

SyntaxError: in
>>> gamestate=[
>>> drawTTTGrid
Traceback (most
File "<stdin>"
NameError: name
>>> pygame.display
>>> backgrounds
>>> gamestate=[
>>> drawTTTGrid
(0,0)=0 s=0 cel
(0,1)=1 s=0 cel
(0,2)=2 s=0 cel
(1,0)=3 s=1 cel
(1,1)=4 s=1 cel
(1,2)=5 s=1 cel
(2,0)=6 s=0 cel
(2,1)=7 s=0 cel
(2,2)=8 s=0 cel
>>> pygame.display.flip()
>>> pygame.event.poll()
<Event(1-ActiveEvent {'gain': 0, 'state': 1})>

```

## Play a stimulus sequence on the 3x3 grid

With the previous code we could draw a single frame of the game-state + stimulus sequence. To play a whole trial's stimuli we need to add a loop running over time, updating the display state as indicated by the loaded stimulus sequence. This is done in 3 parts:

The `playStimulusSequence` function records the start time for the stimulus sequence and then plays the stimuli (looping as needed) until the full stimulus sequence has completed.

```

def playStimulusSequence(stimSeq,stimTime_ms,duration_ms,gamestate=None):
    """play a whole stimulus sequence"""
    t0=client.getTimeStamp() # record starting time
    while client.getTimeStamp()-t0 < duration_ms :
        doSingleFrame(stimSeq,stimTime_ms,t0,gamestate)

```

This function then calls `doSingleFrame` which gets the right frame, draws it on the screen and causes the display to refresh.

```

def doSingleFrame(stimSeq,stimTime_ms,t0,gamestate=None,targetID=None):
    """Draw the appropriate stimulus from the stimSeq if stated at t0"""

```

```

ss = getStimulusState(stimSeq,stimTime_ms,t0)
if len(ss)>9 : ss = ss[:9] # remove unused stimSeq entries
# draw the display
drawTTTGrid(ss,gamestate)
if targetID: print( "*" if ss[targetID-1]>0 else ".",end='',flush=True)
# refresh the display
flip()

```

Finally, as the frame rate of our system may vary, and we want to ensure that stimuli happen at the right time irrespective of this, the `getStimulusState` function uses the time elapsed since the start of the current stimulus sequence to identify which frame should be displayed next.

```

def getStimulusState(stimSeq,stimTime_ms,t0):
    """get the current frame in the stimulus sequence, looping if needed"""
    curtime = client.getTimeStamp()
    loopelapsedtime = curtime - t0
    loopelapsedtime = loopelapsedtime % stimTime_ms[-1] # loop if longer
    than sequence length
    # get the frame we should display = first for which stimTime > curTime
    curframe= [st>loopelapsedtime for st in stimTime_ms].index(True)
    # get the current stimulus state
    return stimSeq[curframe]

```

To test if this all works together, run 1 second of stimulus with:

```

playStimulusSequence(noisecode.stimSeq,noisecode.stimTime_ms,1000)

```

## Add the code to send STIMULUSEVENT information to the Utopia-Hub

The code so-far allows us to load a stimulus sequence and display it to the user. However, this is useless for a BCI unless we can inform the BCI of what the user actually saw. We do this by sending STIMULUSEVENT messages after the display has been re-drawn to inform the recogniser (via the Utopia-HUB) what the current stimulus state was.

As mentioned above, accurate timing of when the display changed is critical for performance of visual-evoked-response BCIs. Thus, we create a time-stamp for the display update as soon as possible after the display has been re-drawn<sup>2</sup>. Note: these timestamps are measured in milliseconds. Here we used the default timestamp clock provided by the utopia-client, however

---

<sup>2</sup> Note: some high precision display frameworks return the *exact* time at which the display refreshed. Using this time-stamp should increase timing performance and possibly BCI performance.

any timestamp clock is usable, so long as it measures in milliseconds and is used consistently for **all** messages originating from **this** client. Note2: for ease of implementation, clients time-stamp clocks **do not** need to be synchronized or have consistent offsets as clock alignment is performed directly in the Utopia-HUB.

First we define the mkStimulusEvent the STIMULUSEVENT message is created based on the current stimulus state. At this point the objectIDs are also added to the message, these are **unsigned 8-bit integers** in the range [1-255] used to uniquely identify every selectable object during stimulation, and it is the presentation systems responsibility to ensure they are unique. As there may be more than one presentation system running at the same time, e.g. when using multiple independent LED buttons, it is important to ensure that there are no objectID clashes. Here we assume we are the only running presentation system running, so use objectIDs 1-9. Note: objectID==0 is **reserved** for indicating the currently cued target in calibration mode (see section [Add information to tell the recogniser what the target is](#)).

```
def mkStimulusEvent(stimulusState,timestamp=None,objIDs=None):
    """make a valid stimulus event for the given stimulus state"""
    if timestamp is None:
        timestamp=client.getTimeStamp()
    if objIDs is None : objIDs = range(1,len(stimulusState))
    return StimulusEvent(timestamp,objIDs,stimulusState)
```

Next, the sendStimulusEvent function actually sends this stimulus event information to the utopiaHUB.

```
def sendStimulusEvent(stimulusState,timestamp=None,objIDs=None):
    """Send a message to the Utopia-HUB informing of the current stimulus state"""
    client.sendMessage(mkStimulusEvent(stimulusState,timestamp,objIDs))
```

Test that this works by sending a fake stimulus event:

```
sendStimulusEvent([0,0,0,1,1,1,0,0,0])
```

If this works you should see the event being received in the fakerecogniser window:

```
***H+
Got Message: t:HEARTBEAT ts:2731111 v:NULL <- server
**Got Message: t:STIMULUSEVENT ts:2731441 v[8]:{1,0}{2,0}{3,0}{4,1}{5,1}{6,1}{7,0}{8,0} <- server
****H
Got Message: t:HEARTBEAT ts:2732194 v:NULL <- server
```

Finally, we modify doSingleFrame to include the sending of the stimulus events describing the current frame:

```
def doSingleFrame(stimSeq,stimTime_ms,t0,gamestate=None):
    """Draw the appropriate stimulus from the stimSeq if stated at t0"""
```

```

ss = getStimulusState(stimSeq,stimTime_ms,t0)
if len(ss)>9 : ss = [int(s) for s in ss[:9]] # remove unused stimSeq
entries
# draw the display
drawTTTGrid(ss,gamestate)
# refresh the display
flip()
# get the flip time
ts = client.getTimeStamp()
# send the stimulus event
sendStimulusEvent(ss,timestamp=ts)

```

You can now run this again as before, it will look the same.

```
playStimulusSequence(noisecode.stimSeq,noisecode.stimTime_ms,1000)
```

But if you look in the fakerecogniser window you will see the stimulus events you have sent are being received:

```

*Got Message: t:STIMULUSEVENT ts:3157804 v[8]:{1,0}{2,0}{3,0}{4,1}{5,0}{6,1}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157808 v[8]:{1,0}{2,0}{3,0}{4,1}{5,0}{6,1}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157809 v[8]:{1,0}{2,0}{3,0}{4,1}{5,0}{6,1}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157812 v[8]:{1,0}{2,0}{3,0}{4,1}{5,0}{6,1}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157813 v[8]:{1,0}{2,0}{3,0}{4,1}{5,0}{6,1}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157815 v[8]:{1,1}{2,0}{3,0}{4,0}{5,1}{6,0}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157816 v[8]:{1,1}{2,0}{3,0}{4,0}{5,1}{6,0}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157824 v[8]:{1,1}{2,0}{3,0}{4,0}{5,1}{6,0}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157825 v[8]:{1,1}{2,0}{3,0}{4,0}{5,1}{6,0}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157827 v[8]:{1,1}{2,0}{3,0}{4,0}{5,1}{6,0}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157828 v[8]:{1,1}{2,0}{3,0}{4,0}{5,1}{6,0}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157829 v[8]:{1,1}{2,0}{3,0}{4,0}{5,1}{6,0}{7,1}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157832 v[8]:{1,0}{2,1}{3,1}{4,1}{5,0}{6,1}{7,0}{8,0} <- server
*Got Message: t:STIMULUSEVENT ts:3157836 v[8]:{1,0}{2,1}{3,1}{4,1}{5,0}{6,1}{7,0}{8,0} <- server
*Got Message: t:STIMULUSEVENT ts:3157838 v[8]:{1,0}{2,1}{3,1}{4,1}{5,0}{6,1}{7,0}{8,0} <- server
*Got Message: t:STIMULUSEVENT ts:3157840 v[8]:{1,0}{2,1}{3,1}{4,1}{5,0}{6,1}{7,0}{8,0} <- server
*Got Message: t:STIMULUSEVENT ts:3157845 v[8]:{1,0}{2,1}{3,1}{4,1}{5,0}{6,1}{7,0}{8,0} <- server
*Got Message: t:STIMULUSEVENT ts:3157847 v[8]:{1,0}{2,1}{3,1}{4,1}{5,0}{6,1}{7,0}{8,0} <- server
*Got Message: t:STIMULUSEVENT ts:3157850 v[8]:{1,1}{2,1}{3,0}{4,0}{5,0}{6,1}{7,0}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157852 v[8]:{1,1}{2,1}{3,0}{4,0}{5,0}{6,1}{7,0}{8,1} <- server
*Got Message: t:STIMULUSEVENT ts:3157857 v[8]:{1,1}{2,1}{3,0}{4,0}{5,0}{6,1}{7,0}{8,1} <- server

```

## Add Calibration mode specific logic

In order to actually make predictions of what the users current target is, the Utopia system needs a machine learning model of how the users brain responses to the visual flicker. Whilst the Utopia system included different ways of learning this model, the most common is to **calibrate** the system to the current user using a calibration phase. During this phase the user is instructed to look at a particular target in turn. This means (assuming the user follows this instruction) the system **knows** what stimulus the brain was responding to, and can then use



[supervised-learning](#) techniques (specifically [Canonical Correlation Analysis](#)) to learn the brain response model of this user.

Adding a calibration mode to the tic-tac-toe game means making a couple of additional changes:

1. We need to send messages to tell the recogniser we are entering/leaving calibration mode.
2. We need to add a **cues** to the stimulus display to tell the user where to look.
3. We need to send additional information to the recogniser to tell it which is the users current target object.
4. Wrap
5. it all up in a loop over targets of fixed duration, sending the additional NewTarget message to tell the recogniser when a flicker sequence has ended and a new one has begun.

Addressing each of these in turn:

### Tell the recogniser we entering/leaving calibration mode

To tell the recogniser we are switching modes in general we use the MODECHANGE events. Specifically to enter Calibration mode we use:

```
client.sendMessage(ModeChange(client.getTimeStamp(),"Calibration.supervised"))
```

And to leave it we use:

```
client.sendMessage(ModeChange(client.getTimeStamp(),"idle"))
```

Note: We use the idle mode as a general mode where the BCI is not active.

Note2: Mode changing may take some time (1-2 seconds) on the Recogniser side. Thus do not change modes too rapidly.

Note3: There is no specific acknowledgement that a mode change has taken place. However, after a mode change has occurred a LOG message will be sent with the current mode information.

### Add Cues to the stimulus display to tell the user where to look.

This can be done in many different ways, such as changing the 'flicker' color for the target letter to, say, green. However, the preferred way in Utopia is to add a pre-flicker 'cueing' phase where the target object is highlighted in **green** before the normal flicker-sequence starts.

Here we do this by simply showing the grid with the target object state set to 'target', and then sleeping until the cueDuration is finished.

```
# cue the user to target 5
tgt=5
cuestimstate=[None]*9
```

```
cuestate[tgt]="cue"  
cueDuration = 1 #set cueduration to 1 second  
drawTTGrid(cuestate)  
time.sleep(cueDuration)  
  
# refresh the display  
flip()
```

To use sleep, you must import time, so add the following to the top of the file

```
import time
```

This should look like:





## Add information to tell the recogniser which object is the target

Again this could be done in many different ways -- such as only sending a single objectID, which is the current 'true-target'. In Utopia however we do this using the objectID==0 which has been **reserved** for exactly this purpose. That is objectID==0 must only be used if that object is **known** to be the users current target object.

Adding this information requires a couple of changes to the previous code.

Firstly, the function which sends the STIMULUSEVENT to the recogniser gets an additional argument to specify which sprite is the 'target', and then adds the objectID==0 with the same state to the sending message:

```
def
mkStimulusEvent(stimulusState,timestamp=None,targetState=None,objIDs=None):
    """make a valid stimulus event for the given stimulus state"""
    if timestamp is None:
        timestamp=getTimeStamp()
    if objIDs is None :
        objIDs = list(range(1,len(stimulusState)+1))
    # insert extra 0 object ID if targetState given
    if not targetState is None :
        objIDs.append(0)
        stimulusState.append(targetState)
    return StimulusEvent(timestamp,objIDs,stimulusState)

def
sendStimulusEvent(stimulusState,timestamp=None,targetState=None,objIDs=None
):
    """Send a message to the Utopia-HUB informing of the current stimulus
state"""

    client.sendMessage(mkStimulusEvent(stimulusState,timestamp,targetState,objI
Ds))
```

Secondly, the current target information must be passed through the flicker sequence display function, so it gets sent to the mkStimulusEvent function. So we add the target as an additional argument to playStimulusSequence,

```
def
playStimulusSequence(stimSeq,stimTime_ms,duration_ms,gamestate=None,targeti
dx=None):
    """play a whole stimulus sequence"""
```

```
t0=client.getTimeStamp() # record starting time
while client.getTimeStamp()-t0 < duration_ms :
    doSingleFrame(stimSeq,stimTime_ms,t0,gamestate,targetidx)
```

And do the same for doSingleFrame and pass it through to the sendStimulusEvent function.

```
def
doSingleFrame(stimSeq,stimTime_ms,t0,gamestate=None,targetidx=None,objIDs=None,sendEvents=True):
    """Draw the appropriate stimulus from the stimSeq if stated at t0"""
    ss = getStimulusState(stimSeq,stimTime_ms,t0)
    if len(ss)>9 : ss = ss[:9] # remove unused stimSeq entries
    # draw the display
    drawTTGrid(ss,gamestate,targetidx=targetidx)
    if targetidx:
        print( "*" if ss[targetidx] and ss[targetidx]>0 else
".",end='',flush=True)
    # refresh the display, without getting pending events to run fast
    flip()
    # get the flip time
    ts = getTimeStamp()
    # send the stimulus event
    if sendEvents:
        targetState = ss[targetidx] if targetidx else None
        sendStimulusEvent(
ss,timestamp=ts,targetState=targetState,objIDs=objIDs)
```

## Wrap it all up in a loop over targets of fixed duration

The final stage is to wrap all these changes into a single function which runs the calibration phase by;

1. generates a sequence of cued targets,
2. loops over these targets
3. first cueing the user
4. Tells the recogniser that this is the start of a new flicker-sequence and
5. then displaying the flicker-sequence

To tell the recogniser that a new flicker-sequence has begun we use the NewTarget message.

This is wrapped in a single function runCalibration:

```
import random
def
```

```

runCalibration(stimSeq,stimTime_ms,numTrials=9,cueDuration=2,trialDuration=
4,interTrialDuration=2):
    """Do the calibration stage of the BCI"""
    # generate the sequence of targets
    tgtSeq = [ random.randint(0,8) for i in range(numTrials) ]
    # tell recogniser to go into calibration mode
    client.sendMessage(
ModeChange(client.getTimeStamp(),"Calibration.supervised"))
    # Instruct the user
    showInstruct(["About to start Calibration","look at the highlighted
target","Presee key when ready"],waitKey=True)
    # put a + on each grid square for testing
    gamestate=["+"]*9
    # loop over targets
    for tri,tgt in enumerate(tgtSeq):
        # tell the Recogniser this is the start of a new trial
        client.sendMessage(NewTarget(client.getTimeStamp()))
        # cue the user
        cuestimstate=['off']*9
        cuestimstate[tgt]="cue"
        drawTTTGrid(cuestimstate,gamestate)
        # refresh the display
        flip()
        time.sleep(cueDuration)
        # play the stimulus
        print("\n%d) tgt=%d"%(tri,tgt))
        playStimulusSequence(
stimSeq,stimTime_ms,gamestate=gamestate,duration_ms=trialDuration*1000,targ
etidx=tgt)
        # wait for the inter-trial
        drawTTTGrid(['off']*9,gamestate)
        # refresh the display
        flip()
        time.sleep(interTrialDuration)
    # tell the recogniser that this is the end of the calibration
    client.sendMessage(ModeChange(client.getTimeStamp(),"idle"))
    # tell the user that calibration has finished
    showInstruct(["That was the end of Calibration","Well
Done!"],waitKey=True)

```

## Testing Calibration Mode

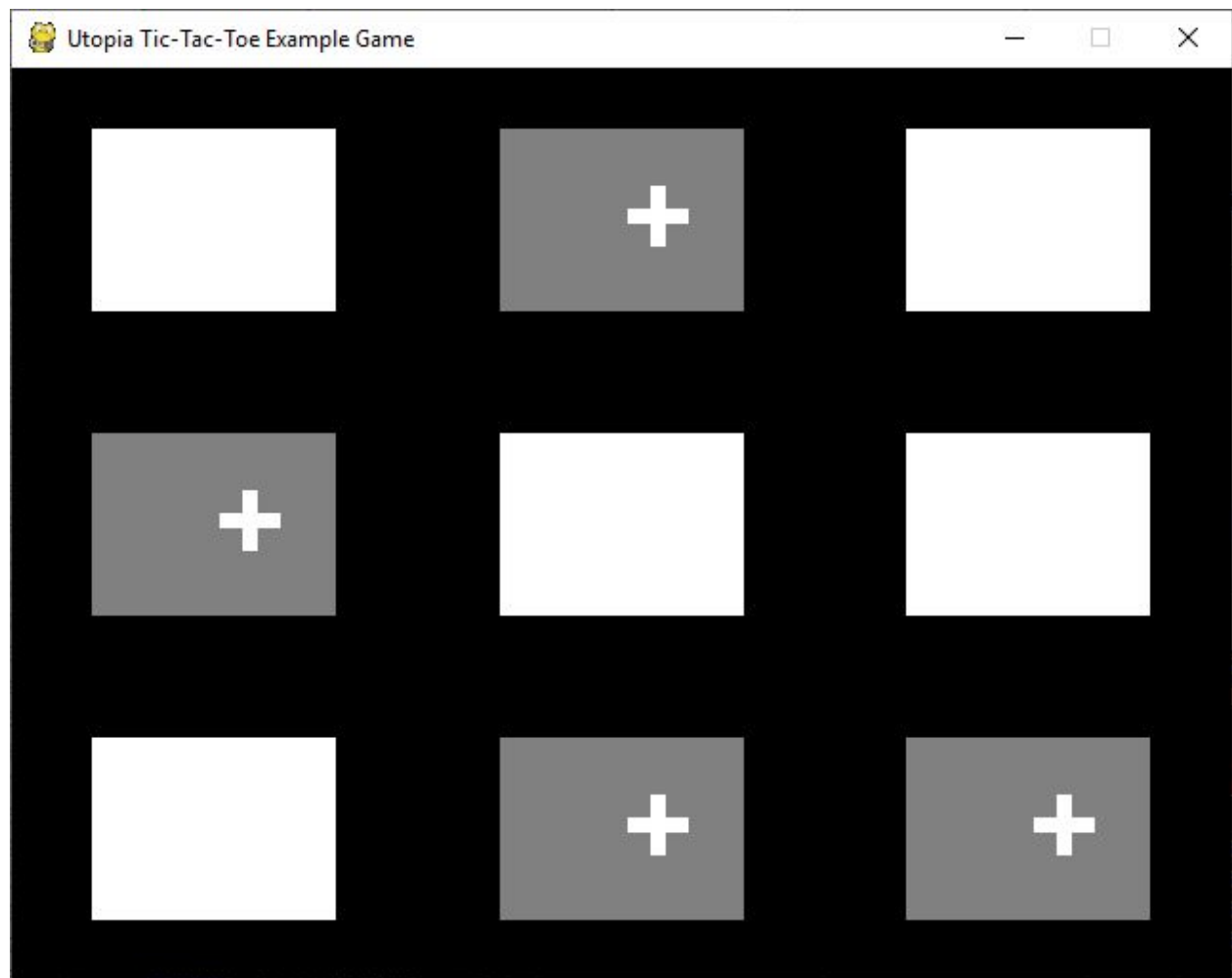
Congratulations! You now have a complete working calibration mode for the tic-tac-toe Utopia BCI. You can now test this mode by adding the following wrapper script to your file:

```
# load the stimulus sequence to use
noise = StimSeq.fromFile("../resources/codebooks/mgold_65_6532_psk_60hz.txt")
stimTime_ms=noise.stimTime_ms
stimSeq=noise.stimSeq

# connect to utopia
client = UtopiaClient()
client.autoconnect()

# run the game
runCalibration(stimSeq,stimTime_ms)
exit()
```

When this runs you should see something like this:



## Add Prediction mode specific logic

After calibration the Utopia system should have learned a model of this user's brain response to the flickering of the tic-tac-toe grid. The next stage is to actually use this response to play the game with your brain!

To do this we need to add the following functionality:

1. code to listen for PREDICTEDTARGETPROB messages
2. Code to select a target when the current prediction is sufficiently confident
3. Code to run the flicker sequence, listen for messages and check for selections, and finally to terminate the flicker sequence when this happens
4. Code to output the selection to the user and update the game state
5. Code to make the AI moves after the computer moves

We discuss the implementation of each of these terms next:

## Listening to the PREDICTEDTARGETPROB messages

The main method to get messages from the utopia-Hub is:

```
# get any messages from the utopia-hub
msgs=client.getNewMessages()
```

This will return all messages received from the hub since the last time it was called.

When no stimulus is currently running this will look something like this:

```
>>> 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']
>>>
```

## Select a target when the current prediction is sufficiently confident

The messages returned from the utopia-HUB are all messages that have been sent to this client. Thus they may contain other messages (for example HEARTBEAT messages) sent from the hub to the client. Thus we should filter these messages to only extract the PREDICTEDTARGETPROB messages. This message then contains 2 fields:

- Yest - the objectID of the predicted target object
- Perr - the estimated probability that the predicted target object is an error

To implement the selection logic we *could* use this error probability along with any contextual information (e.g. the probability of the next letter in written text) to make clever selection decisions.

However, in this case we use the simple selection logic of: *"if the perr is less than .1 then select!"*

Additionally, we would like to *force* a selection at the end of every trial, even if no target was sufficiently good to pass our selection threshold. Thus we return both the result of the selection test, 'isSelected' and the predicted target.

Putting this all together we obtain the following function:

```
def checkForSelectableObject(selectionThreshold=.1):
```

```

    """check if any object prediction is high enough for it to be
    selected"""
    # get any messages with predictions
    msgsgs=client.getNewMessages(0)
    #print("Got %d messages"%(len(msgsgs)))
    # get the predictions and if any is good enough to be selected
    predictedObjID=None
    for msg in msgsgs:
        if msg.msgID==PredictedTargetProb.msgID :
            #print("P:"+str(msg))
            predictedObjID=msg.Yest # record this prediction
            if msg.Perr<selectionThreshold : # good enough to select?
                return (predictedObjID,True)
    return (predictedObjID,False)

```

Run the flicker sequence and terminate early if a selection is made

This can be done in many ways. The most elegant is probably to have the message sending and listening in a separate thread from the main game logic -- so we don't block the game loop. However, in this case, as getting a message is computationally cheap we will just add the listening logic in the main game loop!

This requires writing a new flicker sequence display function which includes the listen to PREDICTEDTARGETPROB messages. Using this function the modified flicker-display loop is:

```

def
playStimulusSequenceWithSelection(stimSeq,stimTime_ms,duration_ms,gamestate
=None,selectionThreshold=.1,objIDs=None):
    """play a whole stimulus sequence"""
    t0=getTimeStamp() # record starting time
    # flush the message queue, so only get predictions after stimulus
    started
    client.getNewMessages(0)
    predictedObjID=None
    isSelected=False
    while getTimeStamp(t0) < duration_ms :
        doSingleFrame( stimSeq,stimTime_ms,t0,gamestate,None,objIDsd)
        curSelObj, isSelected =
checkForSelectableObject(selectionThreshold)
        if curSelObj: # update predicted object
            predictedObjID=curSelObj
        if isSelected: # terminate early if selection passed
            #print("%dms got selection:

```



```
%d"%(getTimestamp(t0),predictedObjID))
    break;
# return last prediction, no matter how bad it was!
return predictedObjID
```

Comparing to the function 'playStimulusSequence' the additional logic is simply to:

1. check if any object is selectable and
2. terminate the flicker sequence early if so
3. Send a selection message to the utopia-hub to inform other clients that an output has been selected

## Output the selection to the user and make the AI moves.

The last step in the prediction mode is to wrap up the single trial predictions into the whole game loop, where we;

1. Send the MODECHANGE messages to indicate start/end of prediction phase
2. Let the user select a move to make
3. Make the move, then have the AI make it's next move
4. Repeat until the end of the game.

Much like the 'runCalibration' function we wrap this up in a single function 'runPrediction' which performs all of the above steps:

```
def
runPrediction(stimSeq,stimTime_ms,numTrials=9,feedbackDuration=2,trialDuration=4,interTrialDuration=2):
    """Do the prediction stage of the BCI"""
    # generate the sequence of targets
    # tell recogniser to go into calibration mode
    client.sendMessage(ModeChange(getTimestamp(),"Prediction.static"))
    # Instruct the user
    showInstruct(["About to start a Game!","look where you want to","put an X","","Press any key to continue"],waitKey=True)
    # loop over targets, user=X, computer=0
    objIDs=list(range(1,9+1))
    gamestate=["+"]*9 # initial game state, no selections
    for tri in range(numTrials):
        # tell the Recogniser this is the start of a new trial
        client.sendMessage(NewTarget(getTimestamp()))

        # play the stimulus
        selectedObjID =
```

```

playStimulusSequenceWithSelection(stimSeq,stimTime_ms,gamestate=gamestate,
duration_ms=trialDuration*1000,selectionThreshold=.1,objIDs=objIDs)

    # update the game state with the selected object, as X from the
    user
    if selectedObjID:
        print("User choose: %d"%(selectedObjID))
        selidx=objIDs.index(selectedObjID)
        gamestate[selidx]='X'
        # give user feedback on their choice
        drawTTTGrid(['feedback' if i==selidx else 'off' for i in
range(9)],gamestate)
        flip()
        waitFor(feedbackDuration*1000)

    # now the computer get to go
    computerObjidx=getComputerMove(gamestate)
    gamestate[computerObjidx]="O"
    print("Computer choose: %d"%(computerObjidx))

    # wait for the inter-trial with updated game state
    drawTTTGrid(['off']*9,gamestate)
    # refresh the display
    flip()
    waitFor(interTrialDuration*1000)
    # tell the recogniser that this is the end of the calibration
    client.sendMessage(ModeChange(getTimeStamp(),"idle"))
    # tell the user that calibration has finished
    showInstruct(["That ends the game","Did you win?","", "Press key to
continue"],waitKey=True,timeout_ms=10000)

```

To add some gameplay, you will need the computer to also make a move therefore, add 'getComputerMove()' to your code

```

# Let the computer make a random move
def getComputerMove(gamestate):
    """get the computer move from the current game state"""
    # just the next available cell
    freesquare=[ s is None or s=="+" for s in gamestate ]
    try:
        moveidx=freesquare.index(True)

```

```
except:
    moveidx=None
    print("Warning: tried to get move in full grid!")
    return moveidx
```

NOTE this move is random and does not make the best selection.

If you run this code via:

```
runPrediction(noisecode.stimSeq,noisecode.stimTime_ms)
```

Then you should see the normal flickering grid, but sometimes the trials will finish early and select X's followed by the computer selection of 0's.

## Testing the tic-tac-toe BCI game

Congratulations, you now have all the bits needed to make a tic-tac-toe BCI game using Utopia. The only thing remaining is to add the final driver method which sets everything up, instructs the user and moves between calibration and prediction phases. This is done with the following code:

```
# load the stimulus sequence to use
noisecode =
StimSeq.fromFile("../resources/codebooks/mgold_65_6532_psk_60hz.txt")
stimTime_ms=noisecode.stimTime_ms
stimSeq=noisecode.stimSeq

# connect to utopia
client = UtopiaClient()
client.autoconnect()

# run the game
runCalibration(stimSeq,stimTime_ms)

while True:
    runPrediction(stimSeq,stimTime_ms)
    # until the user quits
```

You can find the final version of this code in the Utopia-SDK in the file:

python/messagelib/tictactoe.py

## 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

## An easy to use BCI Presentation framework: fakepresentation

Many of the steps used above to make the BCI system are common across **all** stimulus presentation systems, e.g. connecting to the utopia-hub, loading stimulus-sequences, sending STIMULUSEVENT messages after display update, iterating through the stimulus sequence in a time-locked fashion. Thus, to avoid having to copy-paste lots of repeated code, we have made a BCI presentation framework which provides this functionality for any presentation modality, e.g. screen, sounds, LEDs, lasers, etc. Using this framework all you have to do as developer is focus on how the stimulus is presented and the logic of your application. Using this framework the main steps in developing a new BCI presentation component are:

1. Connect to the Utopia-HUB
2. Load the desired stimulus sequence
3. Develop the stimulus display code
4. Add the wrappers for user-calibration and prediction

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

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

winw,winh=window.get_size()

# create set of sprites and add to render batch
batch = pygamelet.graphics.Batch()
background = pygamelet.graphics.OrderedGroup(0)
foreground = pygamelet.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 =
pygamelet.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]=pygamelet.sprite.Sprite(img,x=x+bgoffsetx,y=y+bgoffsety,batch=batch,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,color=(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:

```

#-----
# 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.

## Connect the display to the fakepresentation

```

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 if 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.

```
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,cueDuration=4)

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

## 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 arbitrary 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.