UNSTPB Bucharest, Haskoli Islands

HÁSKÓLI ÍSLANDS

Simon Tactile

Lab Report

Ganea Daria-Marina

Topliceanu Bianca-Andreea

April (2024)

Table of contents:

1 Introduction

The purpose of the „Simon Tactile" Project is to develop another version of the classic application that will use the participant's tactile perception, instead of the vision. For this, we are going to use an Arduino special keyboard where we will input the index of the tactile motors that we want to vibrate, as well as a tactile belt which is connected to an audio interface. By using the computer, the user will feel certain vibrations on the abdomen, where the tactile sensors will be placed. After the execution of the Python programme, they will be asked about the position where they felt the vibrations and will receive a response, stating if they were right or wrong. The final objective of this research project is to draw conclusions regarding the reliability of the human's tactile perception, in comparison with the visual perception.

2 Getting started

2.1 Prerequisites

Before you begin, ensure that you have the following:

- A computer running Python and Arduino;
- Access to a „MADIface x RME Audio" device;
- Access to an Arduino keyboard (we are using a Teensy 3.6);

2.2 Installation

1. Arduino Setup

- Search Arduino IDE on your internet browser and download the latest version available.
- Open the link: https://www.pjrc.com/teensy/td_download.html and follow the steps.
- Install Teensy Loader.
- In Arduino, go to Library Manager and search Teensy, then click and install the library.
- You can now setup the Teensy 3.6 after connecting the keyboard to your USB port. To check if it works, you can test it using a built-in example from the Arduino software: blink, which will basically make the led on the back of the keyboard blink for a while.

2. Python Setup

- Search Python and PyCharm on your internet browser and download the latest versions available. You may also use other Python interpreters than PyCharm. However, this guide will provide you the steps to follow only if you are using PyCharm.
- Open PyCharm -> Settings and make sure that you have the following packages installed:
  numpy, sounddevice, soundfile, keyboard, pyserial, pyusb, setuptools, typing_extensions, Adafruit-Blinka, cffi, pip;

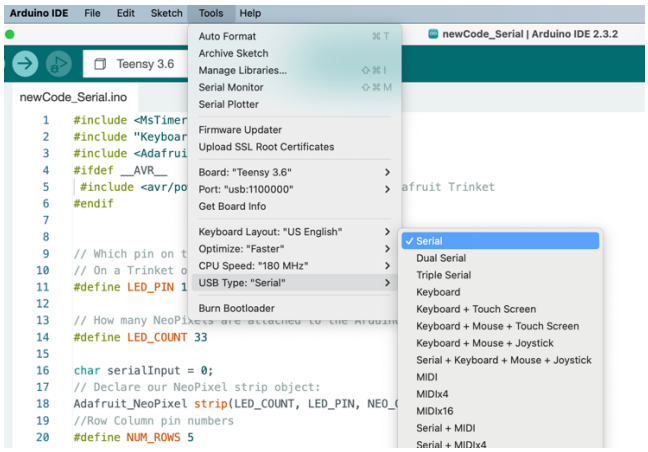
3. MADIface x RME Audio Setup

- Turn on the device. Physically connect the "MADIface xt RME" to your computer, ensuring all the connections are secure.
- If you need specific drivers for the "MADIface xt RME Audio" device, ensure they are downloaded and installed. Check the manufacturer's website for drivers compatible with your operating system.
- Go to the link on github and configurate your workspace in the RME app: https://github.com/Acute-hi-is/Acute-procedures
- The device will be properly connected only if the RME window pops up when you plug in the USB cable. If not, something must be wrong.

3 Methodology

3.1 Arduino

Set the USB Port to be a Serial. We will use this when linking the Arduino code to the Python one.



Arduino Keyboard Layout

This is the keyboard layout as we configurated it in the Arduino code. We are only using the numbers on the keypad for this project.

NOK					
NOK	0	1	2	3	A
NOK	4	5	6	7	B
NOK	8	9	10	11	C
NOK	12	13	14	15	D
NOK	F	G	H	i	E

The NOK column does not physically exist, it is shown here just for code understanding purposes.

Arduino final code
The entire Arduino code, with all the necessary explanations can be found attached at the end of this report.

Some documentation that we have used from github:
https://github.com/Acute-hi-is/Matrix-keyboard

3.2 Python

In the Python code for Windows computers that is attached at the end of the report, you have to modify the 5th line of the code by replacing 'COM3' with the port that your Arduino keyboard is connected to. In order to find it, go to Device Manager. You also need to identify which channel is your audio interface using.

3.3 Final code

In order to run the final application, you have to follow the next steps:

- Turn on the MADIface x RME audio interface and connect it to your computer.
- Connect the keyboard to your computer and make sure that the USB port that you are using is the same as written in your Python code.
- Open the Arduino code and press „Upload". Make sure that your Serial Monitor is not opened.
- Open PyCharm and run the Python code.
- You should now be able to enter the commands for the tactile sensors using the Arduino keyboard. After running the experiment, you should look for a text file called *„simons_tactile.txt"* where the results are stored. Please keep in mind that the results for a participant are automatically erased when you want to run the programme again.

4 Troubleshooting

Hardware Issues: Only a specific number of LEDs work simultaneously on the Arduino Keyboard.

Software Issues:

It is worth mentioning that you are most likely to encounter issues that we were not able to solve if you are using a Macbook when connecting to the MADIface. A Macbook works just fine for testing the code in Arduino, as well as in Python, but only up to the point where we use the audio interface. The main issue here is that the RME appears as an USB device, and not as an audio one, which makes it impossible to continue on a Mac computer.

5 Future improvements

The process of collecting the data can be automatised by sending the data directly into a database. Furthermore, the existing codes should be tested on more people in order to increase the accuracy of the experimental results.

6 Results and conclusions

The results that you can find attached below are for a group of 6 people, divided into 2 subgroups. Each person has received a number of 1 to 10 succesive vibrations as follows: the first n-1 as before, and another one that is completely new. The first subgroup has received feedback after each iteration of the programme telling them if they were right or wrong. The second subgroup did not receive any feedback during the experiment.

Subgroup 1 has shown significantly better results than subgroup 2, which was for sure caused by the received feedback. People in subgroup 2 were much more likely to make mistakes throughout the entire experiment.

Subgroup 1

P1	Vibrations entered	Vibrations Felt	Right / Wrong
1	6	10	Wrong
2	6 **15**	3 **15**	Wrong
3	6 15 **8**	11 3 **8**	Wrong
4	6 15 **8** 3	15 3 **8** 0	Wrong
5	6 15 **8** 3 12	15 3 **8** 7 0	Wrong
6	6 **15 8** 3 12 0	7 **15 8** 3 10 5	Wrong
7	6 **15 8** 3 12 0 3	3 **15 8** 5 2 6 0	Wrong
8	6 **15 8 3** 12 0 3 15	3 **15 8 3** 5 1 0 5	Wrong
9	6 **15 8** 3 12 0 3 15 12	3 **15 8** 11 4 6 9 10 8	Wrong
10	6 **15 8** 3 12 0 3 15 12 6	7 **15 8** 5 9 10 6 2 3 7	Wrong

P2	Vibrations entered	Vibrations Felt	Right / Wrong
1	**6**	**6**	Right
2	**6 15**	**6 15**	Right
3	**6 15** 8	**6 15** 0	Wrong
4	**6 15** 8 3	**6 15** 1 3	Wrong
5	**6 15** 8 3 12	**6 15** 4 13 3	Wrong
6	**6 15** 8 3 12 0	**6 15** 4 12 6 14	Wrong
7	**6 15** 8 3 12 0 3	**6 15** 1 3 13 4 3	Wrong
8	**6 15** 8 3 12 0 3 15	**6 15** 1 3 12 15 0 14	Wrong
9	**6 15** 8 3 12 0 3 15 12	**6 15** 1 13 7 4 13 14 3	Wrong
10	**6 15** 8 3 12 0 3 15 12 6	**6 15** 4 7 8 11 12 2 14 2	Wrong

P3	Vibrations entered	Vibrations Felt	Right / Wrong
1	**6**	**6**	Right
2	**6 15**	**6 15**	Right
3	**6 15 8**	**6 15 8**	Right
4	**6 15 8 3**	**6 15 8 3**	Right
5	**6 15 8 3 12**	**6 15 8 3 12**	Right
6	**6 15 8 3 12 0**	**6 15 8 3 12 0**	Right
7	**6 15 8 3 12 0** 3	**6 15 8 3 12 0** 1	Wrong
8	**6 15 8 3 12 0** 3 15	**6 15 8 3 12 0** 2 5	Wrong
9	**6 15** 8 3 12 **0** 3 15 12	6 15 12 3 0 5 2 1 6	Wrong
10	**6 15** 8 3 12 **0** 3 15 12 6	**6 15** 12 3 12 **0** 6 5 4 2	Wrong

Subgroup 2

P4	Vibrations entered	Vibrations Felt	Right / Wrong
1	5	1	Wrong
2	**5** 13	**5** 10	Wrong
3	**5** 13 11	**5** 9 11	Wrong
4	**5 13 11** 8	**5 13 11** 4	Wrong
5	**5 13 11 8 0**	**5 13 11 8 0**	**Right**
6	**5 13 11 8 0** 15	**5 13 11 8 0** 11	Wrong
7	**5 13 11 8 0** 15 6	**5 13 11 8 0** 11 9	Wrong
8	**5 13 11 8 0** 15 6 4	**5 13 11 8 0** 11 6 9	Wrong
9	**5 13 11 8 0** 15 6 4 12	**5 13 11** 11 8 0 11 6 8	Wrong
10	**5 13 11 8 0** 15 6 4 12 14	**5 13 11 8 0** 11 6 5 4 8	Wrong

P5	Vibrations entered	Vibrations Felt	Right / Wrong
1	5	8	Wrong
2	5 **13**	8 **13**	Wrong
3	5 **13** 11	4 **13** 15	Wrong
4	5 **13** 11 8	4 **13** 15 12	Wrong
5	5 **13** 11 8 **0**	8 **13** 15 12 **0**	Wrong
6	5 **13** 11 8 **0 15**	8 **13** 15 12 **0 15**	Wrong
7	5 **13** 11 8 **0 15** 6	8 **13** 15 12 **0 15** 10	Wrong
8	5 **13** 11 8 **0 15** 6 4	8 **13** 15 12 **0 15** 10 9	Wrong
9	5 **13** 11 8 **0 15** 6 4 12	8 **13** 15 12 **0 15** 10 9 13	Wrong
10	5 **13** 11 8 0 **15** 6 4 **12 14**	8 **13** 15 0 12 **15** 10 9 **12 14**	Wrong

P6	Vibrations entered	Vibrations Felt	Right / Wrong
1	5	0	Wrong
2	5 13	0 4	Wrong
3	5 13 **11**	4 8 **11**	Wrong
4	5 13 **11** 8	4 8 **11** 4	Wrong
5	5 13 **11** 8 0	4 8 **11** 4 8	Wrong
6	5 13 **11** 8 0 15	4 8 **11** 0 12 8	Wrong
7	5 13 **11** 8 0 15 6	4 8 **11** 4 12 11 8	Wrong
8	5 13 **11** 8 0 15 6 4	4 8 **11** 0 6 0 8 11	Wrong
9	5 13 11 8 0 15 6 4 12	0 0 8 3 8 5 0 8 3	Wrong
10	5 13 11 8 0 15 6 4 **12** 14	0 12 15 12 3 6 10 0 **12** 15	Wrong

7 Arduino and Python Codes

Arduino final code (SimonsTactile_arduinoCode.ino)

```
#include <MsTimer2.h>
#include "Keyboard.h"
#include <Adafruit_NeoPixel.h>
#ifdef __AVR__
#include <avr/power.h> // Required for 16 MHz Adafruit Trinket
#endif
// Which pin on the Arduino is connected to the NeoPixels?
// On a Trinket or Gemma we suggest changing this to 1:
#define LED_PIN 19
// How many NeoPixels are attached to the Arduino?
#define LED_COUNT 33

char serialInput = 0;
// Declare our NeoPixel strip object:
Adafruit_NeoPixel strip(LED_COUNT, LED_PIN, NEO_GRBW + NEO_KHZ800);
//Row Column pin numbers
#define NUM_ROWS 5
#define NUM_COLS 7
//number of iterations of identical keyscan values before we trigger a keypress
#define DEBOUNCE_ITER 5
//milliseconds between each scan. SCAN_PERIOD * DEBOUNCE_ITER = minimum response time
#define SCAN_PERIOD 3
#define KEY_RELEASED 1
#define KEY_PRESSED 0
#define KEY_FUNCTION -1
#define NOK 0
#define KC_NONUS_BACKSLASH        ( 100  | 0xF000 )
#define KC_NONUS_HASH             (  50  | 0xF000 )
byte rowPins[NUM_ROWS] = {26,28,30,32,37}; // the pins where the rows are connected in
order
byte colPins[NUM_COLS] = {1,2,7,22,21,10,12}; // the pins where the columns are connected
in order
byte keyIterCount[NUM_ROWS][NUM_COLS];
byte keyState[NUM_ROWS][NUM_COLS];

// new keymap; when pressing the buttons on the keyboard, these are the assigned values for
each button
// the NOK keys do not exist, are put here just to prevent errors
int keyMap[NUM_ROWS][NUM_COLS] = {
        {NOK, NOK, 0,1,2,3,'A'},
        {NOK, NOK, 4,5, 6, 7,'B'},
        {NOK, NOK, 8, 9, 10, 11,'C'},
        {NOK, NOK, 12, 13, 14, 15,'D'},
        {NOK, NOK, 'F', 'G', 'H', 'I', 'E'}
};
```

```
// what does this do? nothing. It's like a shift but does not work anyway
int funcKeyMap[NUM_ROWS][NUM_COLS] = {
         {NOK, NOK, 0,1,2,3,'A'},
        {NOK, NOK, 4,5, 6, 7,'B'},
        {NOK, NOK, 8, 9, 10, 11,'C'},
        {NOK, NOK, 12, 13, 14, 15,'D'},
        {NOK, NOK, 'F', 'G', 'H', 'I', 'E'}
};
// how are the leds numbered, from right corner
int LedMAP[NUM_ROWS][NUM_COLS] = {
        {NOK, NOK, 24,23,22,21,20},
        {NOK, NOK, 15,16,17,18,19},
        {NOK, NOK, 14,13,12,11,10},
        {NOK, NOK, 5,6,7,8,9},
        {NOK, NOK, 4,3,2,1,0}
};

boolean funcMode = false;
void setup()
{
  Serial.begin(9600);
  strip.begin();           // INITIALIZE NeoPixel strip object (REQUIRED)
  strip.show();            // Turn OFF all pixels ASAP
  strip.setBrightness(100); // Set BRIGHTNESS to about 1/5 (max = 255)

  for (int row = 0; row < NUM_ROWS; row++)
  {
    pinMode(rowPins[row], INPUT);
  }

  for (int col = 0; col < NUM_COLS; col++)
  {
    pinMode(colPins[col], INPUT_PULLUP);
  }
// Set the initial values on the iter count and state arrays.
  for (int row = 0; row < NUM_ROWS; row++)
  {
    for (int col = 0; col < NUM_COLS; col++)
    {
// Initial iter value is debounce + 1 so that a key transition isn't immediately detected
on startup.
      keyIterCount[row][col] = DEBOUNCE_ITER + 1;
      keyState[row][col] = KEY_RELEASED;
    }
  }
  LEDBOX();
  //Keyboard.begin();
  MsTimer2::set(SCAN_PERIOD, keyScan);
  MsTimer2::start();
```

```
}
// Pressing the FN key could potentially shift the scan code between the key being pressed
// and being released. If the FN key is hit then any pressed keys have to be reset to be in
the
// 'released' state and their iter counts set to DEBOUNCE_ITER+1.
// Quick improvement: Only do this if the scan codes are different in the two maps. This
means that
// any keys that are the same between the layers like the modifiers will remain pressed.
void resetKeyStates(bool funcMode)
{
  // Set the initial values on the iter count and state arrays.
  for (int row = 0; row < NUM_ROWS; row++)
  {
    for (int col = 0; col < NUM_COLS; col++)
    {
      // Only reset if they're different on the two layers.
      if (keyMap[row][col] != funcKeyMap[row][col])
      {
        keyIterCount[row][col] = DEBOUNCE_ITER + 1;
        // If it's PRESSED, then send information over the serial port
        if (keyState[row][col] == KEY_PRESSED)
        {
          // Indicate that it's a key release on the previous map code
          Serial.print("R");
          Serial.print(funcMode ? funcKeyMap[row][col] : keyMap[row][col]);
          Serial.println();
          // Indicate that it's a key press on the new map code
          Serial.print("P");
          Serial.print(funcMode ? keyMap[row][col] : funcKeyMap[row][col]);
          Serial.println();
        }
      }
    }
  }
}
// we have a debounced key transition event, either pressed or released.
void transitionHandler(int state, boolean fnMode, int row, int col)
{
  // Pick which keyMap we're using based on whether we're in func mode
  int scanCode = fnMode ? funcKeyMap[row][col] : keyMap[row][col];
  if (state == KEY_PRESSED)
  {
    // Send the scanCode over the serial port when a key is pressed
    // Serial.print("P"); // Indicate that it's a key press
    Serial.print(scanCode);
    Serial.println();
  }

}
```

```
void colorWipe(uint32_t color, int wait)
{
  for(int i=0; i<strip.numPixels(); i++)
  { // For each pixel in strip...
    strip.setPixelColor(i, color);         //  Set pixel's color (in RAM)
    strip.show();                          //  Update strip to match
    delay(wait);                           //  Pause for a moment
  }
}

// this is for changing the color when pressing the button
// however, we don't want to change the color for the purpose of the experiment
void setLEDRoutine(int state, int row, int col)
{
    //strip.clear();
    strip.setPixelColor(LedMAP[row][col], strip.Color(255,   255,   255));
    strip.show();
}

void LEDBOX (void)
{
  int LEDNumber = 16;
  // the BOX matrix is used to select the LEDs that will be light up
  // only the LEDs that are coloured will be used to input commands in the experiment
  int BOX[] = {
          24, 23, 22, 21,/* 20,*/
          15,16, 17,18,/* 19,*/
         14, 13, 12,  11,/* 10,*/
           5, 6,  7 , 8, /*  9,*/
          /*4,  3,  2,  1,   0*/
          };
  strip.clear();
  for (int Counter = 0; Counter < LEDNumber; Counter++)
  {
    strip.setPixelColor(BOX[Counter], strip.Color(255,255,255)); // BOX[Counter]= the
selected LED, strip.Color(R, G, B)
    strip.show();
  }
strip.show(); }
void keyScan()
{
  //set the scanFuncMode to false so we can check for any of the FN keys being pressed
  bool scanFuncMode = false;
  //First loop runs through each of the rows,
  for (int row=0; row < NUM_ROWS; row++)
  {
        //for each row pin, set to output LOW
        pinMode(rowPins[row], OUTPUT);
```

```
        digitalWrite(rowPins[row], LOW);
        //now iterate through each of the columns, set to input_pullup,
        //the Row is output and low, and we have input pullup on the column pins,
        //so a '1' is an un pressed switch, and a '0' is a pressed switch.
        for (int col=0; col < NUM_COLS; col++)
        {
            byte value = digitalRead(colPins[col]);
            //if the value is different from the stored value, then reset the count and set
the stored value.
            if(value == KEY_PRESSED && keyState[row][col] == KEY_RELEASED)
            {
              keyState[row][col] = KEY_PRESSED;
              keyIterCount[row][col] = 0;
              LEDBOX();
              setLEDRoutine(LedMAP[row][col], row, col);

            }
            else
              if (value == KEY_RELEASED && keyState[row][col] == KEY_PRESSED)
              {
               keyState[row][col] = KEY_RELEASED;
               keyIterCount[row][col] = 0;
              }
              else
              {
// Stored value is the same as the current value, this is where our debounce magic happens.
// If the keyIterCount < debounce iter then increment the keyIterCount and move on
// if it's == debounce iter then trigger the key & increment it so the trigger doesn't
happen again.
// if it's > debounce iter then we do nothing, except check for the FN key being pressed.
                if(keyIterCount[row][col] < DEBOUNCE_ITER)
                {
                   keyIterCount[row][col] ++;
                }
                else
                   if (keyIterCount[row][col] == DEBOUNCE_ITER)
                  {
                     keyIterCount[row][col] ++;
                    transitionHandler(keyState[row][col], funcMode, row, col);
                  }
                  else
                  {
                //We'll check for the func activation here, we just want to check if
                //any of them are pressed. If any are then set the scan func var to
                //true. It starts as false so if there aren't any func keys pressed
                //it'll remain as false. At the end of the scan we just set the global
                //func bool to whatever the scan local one is.

                //we'll just check the base layer to avoid in sticky situations
```

```
                    if(keyMap[row][col] == KEY_FUNCTION && keyState[row][col] ==
KEY_PRESSED)
                    {
                        scanFuncMode = true;
                    }
                  }
            }
        }
        //now just reset the pin mode (effectively disabling it)
        pinMode(rowPins[row], INPUT);
  }
// lastly if the scanFuncMode is different from the global func mode, then do the keystate
reset,
// and set the global func mode to the scan func mode.
  if(funcMode != scanFuncMode) {
    resetKeyStates(funcMode);
    funcMode = scanFuncMode;
  }
}

// we use this piece of code to be able to write the data in pyhton console
void loop() {
  if (Serial.available())
  {
    char c = Serial.read();
    Serial.print(c);
  }
}
```

Python Code

```
import serial
# Configure serial port
ser = serial.Serial('COM3', 9600)

# Save commands to a text file
file_name = "simons_tactile.txt"

# Read an array of integers from the keyboard, one element at a time
print("How many sensors do you want to vibrate?")
n = int(ser.readline().decode().strip())
print(n)
array = []

print("Enter the elements one by one:")
for _ in range(n):
    element = int(ser.readline().decode().strip())
    array.append(element)
    print(element)

print("Commands read from keyboard:", array)

# ivan test code

# Finnur Pind 18.03.2021
# Pilot test for audio-tactile experiment using the L5 display
# !/usr/bin/env python3
"""Play a sine signal."""
import argparse
import sys
import numpy as np
import sounddevice as sd
import time
import msvcrt
import keyboard as kb


########## READ INPUT DATA ##########
# Mainly which audio device to use...
def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
```

```
    parents=[parser])
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='output device (numeric ID or substring)')
args = parser.parse_args(remaining)

f = 0
a = 0
start_idx = 0


############################################################


# Playback function
def callback(outdata, frames, time, status):
    if status:
        print(status, file=sys.stderr)
    global start_idx
    t = (start_idx + np.arange(frames)) / samplerate
    t = t.reshape(-1, 1)  # Change from row vector to column vector?

    # To play the same sine wave to both output channels use this:
    outdata[:] = a * np.sin(2 * np.pi * f * t)  # Compute the sine
wave
    start_idx += frames


samplerate = sd.query_devices(args.device,
'output')['default_samplerate']

# Close the database connection

########## EXPERIMENT PARAMETERS #########
##CHANGE EVERY TIME
participantID = 1

nTrials = 4  # Number of iterations per condition, should be
divisible by 4 (the code assumes 4 blocks of trials, i.e. 3 breaks
nTestTrials = 1  # Number of iterations per condition
doStartupTest = 1  # Play a signal through all motors to make sure
everything is working
stimuliDuration = 0.2  # duration of stimulis (in secs)
delayBetweenStimuli = 0.05  # delay between stimuli 1 and stimuli 2
responseTimeWindow = 5  # time window for response
delayAfterResponse = 0.5  # time after response is given until next
is played
##########################################


if doStartupTest:
    print('Running startup test...')
    nCoils = 32
    # all_channels = list(range(nCoils))
    f = 100
    a = 0.1
    i = 0
    noMotors = int(n-1)
    # we want n motors to vibrate separately, not altogether
```

```
if doStartupTest:
    print('Running startup test...')
    nCoils = 32
    # all_channels = list(range(nCoils))
    f = 100
    a = 0.1
    i = 0
    noMotors = int(n-1)
    # we want n motors to vibrate separately, not altogether


for element in array:
    print('Startup test, motor:', element)

    asio_ch = sd.AsioSettings(channel_selectors=[element])
    # start_idx = 0

    with sd.OutputStream(device=8, channels=1, callback=callback,
                         samplerate=samplerate,
extra_settings=asio_ch):
        time.sleep(1)
        sd.stop()
file_name = "simons_tactile.txt"

# Read an array of integers from the keyboard, one element at a time
print("Where did you feel the vibrations? Please enter them in
order.")
arrayResults= []

print("Enter the elements one by one:")
for _ in range(n):
    element2 = int(ser.readline().decode().strip())
    arrayResults.append(element2)
    print(element2)

print("You felt the vibrations in the following order:",
arrayResults)


result = array == arrayResults
if (result == True):
    print ('You are right!')
if (result == False):
    print ('You are wrong!')

with open(file_name, 'w') as file:
    file.write("Commands read from keyboard:\n")
    for element in array:
        file.write(f"{element}\n")
    file.write("Vibrations felt by the user:\n")
    for element2 in arrayResults:
        file.write(f"{element2}\n")
    file.write(f"{result}\n")
print(f"Results have been saved to {file_name}")
```