UNIVERSITÄT KOBLENZ · LANDAU
Faculty 4: Computer Science

WeST People and Knowledge Networks
Institute for Web Science and Technologies

Enhancing Code Navigation by Integrating Voice and Gaze

Master's Thesis

in partial fulfillment of the requirements for
the degree of Master of Science (M.Sc.)
in Informatik

submitted by
Pooya Oladazimi

First supervisor: Prof. Dr. Steffen Staab
Institute for Parallel and Distributed Systems,
University of Stuttgart

Second supervisor: Korok Sengupta
Institute for Parallel and Distributed Systems,
University of Stuttgart

Koblenz, November 2020

Statement

I hereby certify that this thesis has been composed by me and is based on my own work, that I did not use any further resources than specified – in particular no references unmentioned in the reference section – and that I did not submit this thesis to another examination before. The paper submission is identical to the submitted electronic version.

	Yes	No
I agree to have this thesis published in the library.	☐	☐
I agree to have this thesis published on the Web.	☐	☐
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	☐	☐
The source code is available under a GNU General Public License (GPLv3).	☐	☐
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	☐	☐

koblenz, 16.11.2020 [illegible]

(Place, Date) (Signature)

Note

- If you would like us to contact you for the graduation ceremony,

  please provide your personal E-mail address: ....
- If you would like us to send you an invite to join the WeST Alumni

  and Members group on LinkedIn, please provide your LinkedIn ID : ....

Acknowledgement

I would like to express my gratitude to my thesis advisors Prof. Dr. Steffen Staab and Korok Sengupta for their support and supervision during this Master Thesis. Korok Sengupta supported and mentored me from the start of this Thesis to the end which played a big role in completion of my Thesis. This Thesis was a great opportunity for me to learn and challenge myself to accomplish a project.

I am also grateful to the participants who patiently participated in my work showed exceptional calmness and composure.

At the end, I thank my family without whom this success would not have been possible. Thank you.

Abstract

This master thesis tries to investigate the integration of voice and gaze to traditional mouse-keyboard for code navigation. Previous studies in this field suffer from a lack of proper evaluation. The scalability of such systems for supporting more navigational actions is not also clear. Plus, there is a gap in online navigational actions in terms of interaction. We implemented three Jupyter Notebook extensions to overcome the research gaps in this field of study. Our results show that voice is a natural modality for adding to the traditional approach for code navigation, especially for online navigational actions. Besides, we present a prediction system approach that could reduce the transcription error's effect on source code navigation. Our results show that there is a trade-off between accuracy and time for gaze modality. Moreover, we show that we can have a line-wise selection with a coloring approach for gaze instead of region-wise selection. At last, we mention some lessons regarding designing a proper Multimodality approach for code navigation.

Contents

1. Introduction

Code navigation is an essential programming activity, especially during code maintenance and understanding [7, 12, 14, 18, 23]. Programmers traditionally use mouse-keyboard for code navigation. Researchers try to find alternative interaction ways for the traditional mouse-keyboard approach to facilitate code navigation. Alternative interaction techniques utilize Unimodality such as Gaze [9, 29] or a Multimodality such as Voice-Gaze [22]. Despite the efforts, there are some limitations in this field of study. Gaze-based approaches decrease accuracy because of Midas Touch [29]. Voice-based approaches suffer from transcription errors [27]. Additionally, the performance of Multimodality approaches in code navigation is not well studied, especially compared to the mouse-keyboard approach [9, 22, 25]. The latter is important for programmers who are often skilled in using mouse-keyboard. Finally, there is a gap in research regarding online search in terms of interaction during programming. In this study, we aim to overcome the limitations and gaps in using alternative interaction approaches for code navigation.

Rosenblatt et al. [27] clustered code interactions to : (i) Navigational, (ii) Additional, (iii) Removal, and (iv) Selection. Code navigation is important among the mentioned interaction types where a developer browse a set of desire code parts [18]. For instance, A study on software maintenance showed that programmers spend 35% of their time on code navigation [12]. We can group code navigation to offline and online ones. Offline refers to the navigation between identifiers inside the target source code. For example, a study stated that developers often have to trace a variable from declaration to the last usage (or vice versa) to understand a code objective [7]. Besides, foraging for relevant code-related information occupies half of the developers' time during debugging tasks [23]. Online navigation refers to browsing for code snippets and API documentation from online sources. A study showed that developers prefer the coding with examples approach, which means they often look for small code blocks from online sources to copy/paste into their source code to find the best one [11].

Code navigation has been traditionally dependent on keyboard and mouse [27]. Moreover, developers utilize an Integrated Development Environment (IDE) menu for code navigating [20]. Using an IDE for code-related activities increases developers' work load. For instance, one study [17] shows that developers spend, on average, 14% of their time struggling with IDE User Interface to find proper code-related functionalities. On the other hand, there are some issues with the traditional mouse-keyboard approach. The traditional approach might not be suitable for having more complex navigational interactions that demand multiple micro-interactions. For example, when a user tries to find the last modification of a variable, this has to be done by reading and scrolling, which adds navigational overhead to programming [12]. Moreover, interaction with code has not evolved with the change in hardware

[29]. For instance, when hands are busy in one task, no other means of interaction are available to perform some other task on the code. For example, if a developer wants to look at a function body while writing code documentation, she/he has to stop writing and move to the function body with the mouse-keyboard, and then come back to continue the previous process. The mentioned manner could cause a distraction in the workflow and hamper multi-tasking capabilities.

Researchers investigated other interaction methods as alternatives for mouse and keyboard to overcome traditional code navigating approaches' limitations. The first group of alternative approaches is Unimodality ones, which try to replace mouse-keyboard with a new modality such as gaze [29] or voice [27]. The second group is Multimodality approaches that try to replace mouse-keyboard with a combination of modalities such as voice-gaze [22] or gaze-keyboard [9].

Alternative Unimoadlity approaches empower disabled people who have issues with using mouse-keyboard to engage more in programming [27]. However, they have some limitations for non-disabled programmers. Using only one modality exposes users to all limitations of that specific modality. Gaze-only approaches decrease the code navigation accuracy because of the Midas Touch (unintentional click) issue [9, 29]. Also, using Gaze for clicking adds time overhead to users' actions due to dwelling [9]. On the other hand, transcription error is a common issue with all Voice-based system. Therefore, Voice-only approaches that rely entirely on voice also decrease accuracy. In addition, performing micro cursor interactions (such as moving the cursor up/down) might not be efficient in Voice-only approaches compare to traditional mouse-keyboard [27].

Alternative Multimodality approaches benefit from a combination of modalities that resolve some issues with Unimodality approaches. For instance, users can utilize a switch or key to reduce the Midas Touch issue with Gaze-based approaches [9, 22, 25]. However, alternative Multimodalities also have limitations. The capacity of Multimodality approaches to replace mouse-keyboard in code navigation is not well studied. Some approaches do not have any evaluation [25]. Others only focus on system usability [9, 22]. Consequently, the system performances compare to other approaches (such as traditional mouse-keyboard) are unknown. For instance, although using the Gaze-Voice approach [22] shows positive usability and help disabled programmers, it might slow down non-disabled programmers compare to traditional mouse-keyboard.

Along with the limitations of Unimodality and Multimodality alternative approaches, there are some general limitations in this field of study. The navigational actions supported by some previous works are limited to primitive actions such as going to a code line or moving the cursor. Therefore, the systems' capacity for having more complex navigational actions is not clear. Additionally, there is no study on online navigation (search for code snippets and online documentations) in terms of interaction. The studies on online navigation mainly focus on information extraction in the shape of recommendation systems [24, 26], which is out of the boundary of this master thesis. Moreover, the alternative approaches performance compare to traditional mouse-keyboard are not well studied. This is particularly important for non-disabled programmers who traditionally rely on mouse-keyboard for code-related activities.

We try to overcome the previous approaches' gaps and limitations by integrating a new modality to the traditional mouse-keyboard approach. The goal is to study the natural way of having an additional modality to existing mouse or keyboard approaches. Also, we include new navigational actions (such as online search) to study approaches scaleability to support more navigational actions. Moreover, we try to evaluate the new approaches performance with compare to traditional mouse-keyboard. To achieve the goals, we implement three Jupyter Notebook[1] extensions, where participants perform navigational actions with Gaze and Keyboard, Voice and Mouse, and traditional Mouse-Keyboard Multimodalties. For Gaze-Keyboard setup, we adapt CodeGazer [29] coloring approach. We also use the Keyboard as the mechanical switch like Voiceye [22] to decrease Midas Touch. We utilize a keyboard to enter online search queries since gaze typing could decrease users' speed [13, 16]. For Voice-Mouse setup, we use voice for navigational commands like similar voice-based studies [22, 27]. We use Mouse for primitive micro-interactions such as moving cursor and scrolling. In the end, We evaluate our system in both controlled and uncontrolled experiment environments for summative and formative evaluation. We also try to find the design challenges for each approach.

[1]`https://jupyter.org/`

2. Background

Offline navigation for finding code parts is an important part of code implementation. A study on code maintenance [12] showed that developers often start with gathering relevant code sections and navigating through them to understand the code. Also, jumping from a method call to the method body or from variable usage to declaration form more than half of code navigation activities. Moreover, developers often jump back to the point which they left for navigation. Another study [8] finds out that navigation for understanding code is an essential activity in code modification. Besides, Navigation by jumping directly from one element to another (such as from method call to body) or by the search are common navigation types. A study on Visual Studio usage showed that navigation is essential for code understanding [5].

Online navigation, where a developer tries to find a code snippet or API documentation on the internet, is another navigation type. A study on developers' searching behaviors showed that searching is one of the most common activities during programming, which directly impacts a developer's performance [28]. Besides, developers mostly search while performing a code change activity such as code understanding or finding usage examples [28]. Ohta et al. [21] investigated the programmer's Copy/Paste behavior during software development. According to their study, (i) locating the target code snippets, (ii) Coping it, (iii) Pasting it to source code, and (iv) editing the code snippet are the four steps in reusing a code snippet. Their results also showed that code reusing by Copy/Paste is ubiquitous among software developers. In an attempt to design a recommendation system, Iqbal et al.[11] showed that code reusability with Copy/Paste is a common practice among software developers these days.

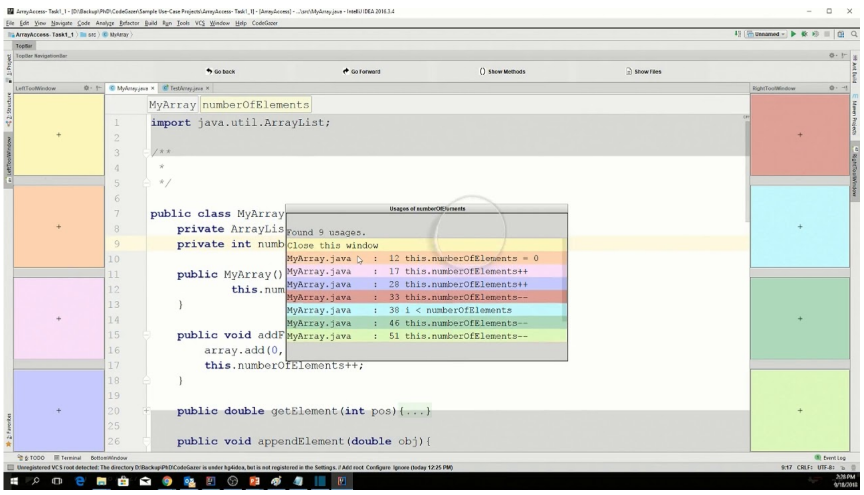
In the next subsections, we describe the different approaches undertaken to facilitate code navigation. These studies have two common characteristics. First, they all try to introduce alternatives for the traditional mouse-keyboard approach. Second, they only focus on offline navigation. According to our literature review, we have not come across any study that investigates online navigation. Research on online navigation has been mostly in the domain of recommendation systems [24, 26], which we consider out of the boundary of this master thesis. We group the related systems into two subsections. (i) unimodality approaches that use one modality for code navigation. (ii) multimodality approaches that use a combination of different modalities (such as gaze + keyboard).

2.1. Unimodality

Alternative Unimodality approaches aim to study the feasibility of replacing mouse-keyboard with another modality like gaze or voice. This is especially crucial for disabled programmers who have issues using mouse and keyboard.

CodeGazer is an IntelliJIDE plugin that aims to study gaze-based source code navigation [29]. They try to compare gaze-based source code navigation with mouse and keyboard. CodeGazer utilizes the ActiGaze [15] coloring approach for clicking (Figure 1). Besides, users could perform scrolling by dwelling at the top-bottom part of the screen. The study results show that gaze-based source code navigation is comparable with keyboard-based navigation but slower than mouse-based. However, some of their participants state that they prefer gaze-based source code navigation cause it is more natural than mouse-keyboard. Also, they find out gaze-based navigation is the best for the Go-To type of actions, where users jump to the function body from the function call and then jump back to the function call.

CodeGazer has some limitations. First, scrolling in CodeGazer suffers from Midas Touch. Participants made unintentional scrolling by looking at the top-bottom part of the screen. The latter might exist because of the CodeGazer's low dwelling time (300ms). Moreover, the study results also show that users commit more mistakes with CodeGazer than the mouse and keyboard approaches, which might be related to the low dwelling time in CodeGazer. More, Codegazer has region-based target selection that means it colors all the identifiers inside a region, which contains several lines (part B in Figure 2). So it not possible to directly selects an identifier in a specific line. This is because the lines are close to each other, and eye trackers are not precise enough to select a specific line.

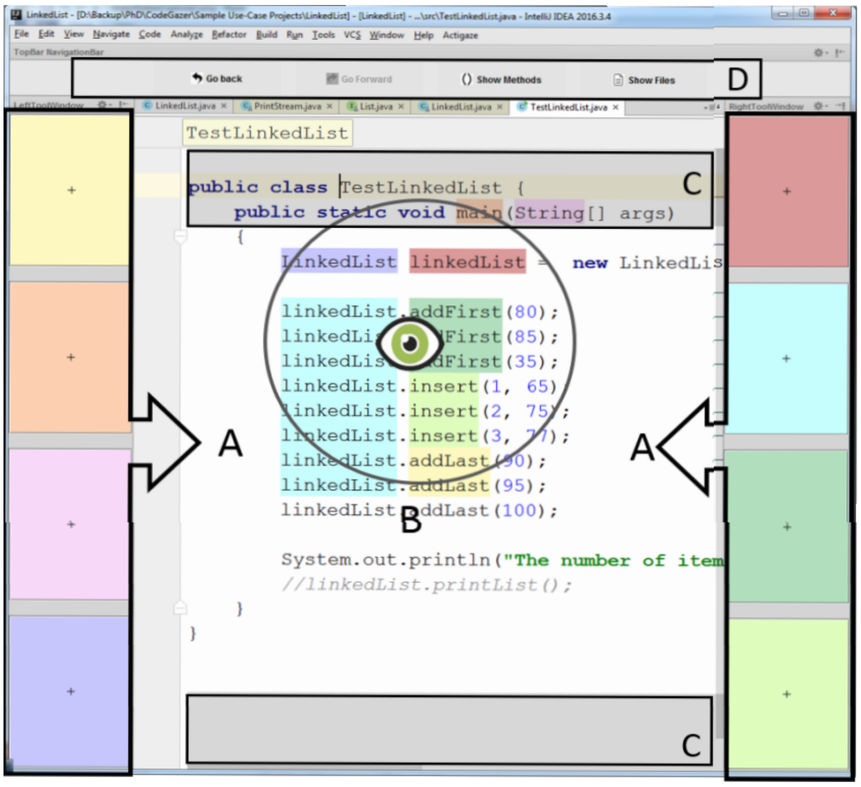

ArrayAccess- Task1_1 - [D:\Backup\PhD\CodeGazer\Sample Use-Case Projects\ArrayAccess- Task1_1] - [ArrayAccess] - ...\src\MyArray.java - IntelliJ IDEA 2016.3.4
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help CodeGazer
ArrayAccess- Task1_1 src MyArray
TopBar
TopBar NavigationBar
Go back
Go Forward
Show Methods
Show Files
LeftToolWindow
MyArray.java
TestArray.java
RightToolWindow
MyArray numberOfElements
Usages of numberOfElements
TODO
Terminal
BottomWindow
Event Log

```
import java.util.ArrayList;

/**
 *
 */

public class MyArray
    private ArrayLis
    private int numb

    public MyArray()
           this.num
    }

    public void addF
        array.add(0,
        this.numberOfElements++;
    }

    public double getElement(int pos){...}

    public void appendElement(double obj){
```

```
Found 9 usages.
Close this window
MyArray.java   :  12 this.numberOfElements = 0
MyArray.java   :  17 this.numberOfElements++
MyArray.java   :  28 this.numberOfElements++
MyArray.java   :  33 this.numberOfElements--
MyArray.java   :  38 i < numberOfElements
MyArray.java   :  46 this.numberOfElements--
MyArray.java   :  51 this.numberOfElements--
```

Unregistered VCS root detected: The directory D:\Backup\PhD\CodeGazer is under hg4idea, but is not registered in the Settings. // Add root Configure Ignore (today 12:25 PM)

Figure 1: CodeGazer. Users can navigate the code by dwelling at side colors.


LinkedList - [D:\Backup\PhD\CodeGazer\Sample Use-Case Projects\LinkedList] - [LinkedList] - ...\src\TestLinkedList.java - IntelliJ IDEA 2016.3.4
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help Actigaze
TopBar NavigationBar
Go back
Go Forward
Show Methods
Show Files
D
LeftToolWindow
LinkedList.java
PrintStream.java
List.java
LinkedList.java
TestLinkedList.java
RightToolWindow
TestLinkedList
C
A
A
B
C

```
public class TestLinkedList {
    public static void main(String[] args)
    {
        LinkedList linkedList = new LinkedLis

        linkedList.addFirst(80);
        linkedLi   First(85);
        linkedLis  dFirst(35);
        linkedList.insert(1, 65);
        linkedList.insert(2, 75);
        linkedList.insert(3, 77);
        linkedList.addLast(90);
        linkedList.addLast(95);
        linkedList  ddLast(100);

        System.out.println("The number of item
        //linkedList.printList();
    }
}
```

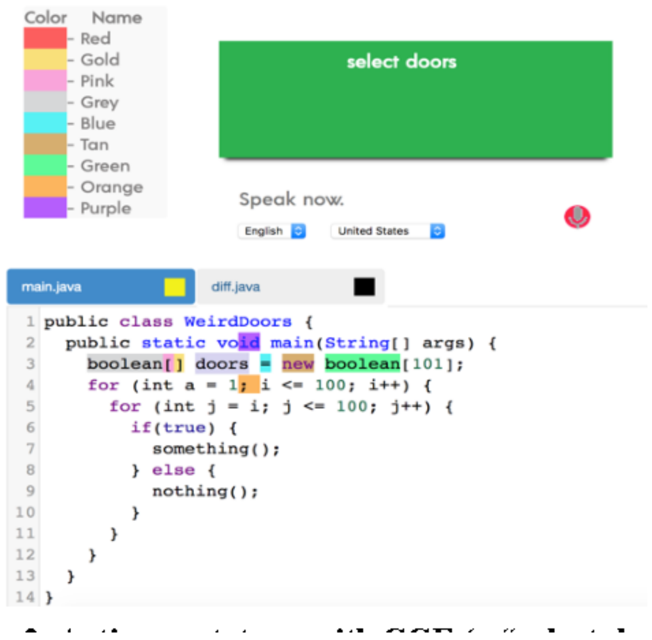
Figure 2: Codegazer. it uses region-based selection (part B). This is is because lines are too much close to each other which makes it difficult to select an specific line directly.

Rosenblatt et al. [27] study voice usage to help people with limited dexterity in code writing and editing. They implemented a prototype VocalIDE[2] for coding in Java with the help of the voice. VocalIDE supports Text Entry, Navigation, Text Selection, Replacement, Deletion, Undo, and Smart Snippet with vocal commands. Also, they use a coloring approach for facilitating code navigation. Their coloring approach gives nine colors to the cursor surrounding context. Instead of using multiple Left and Right vocal commands to move the cursor, users can utilize colors for moving the cursor faster and more comfortable (Figure 3). In addition, users can say the target identifier's name in order to select it. In the case of multiple matches, VocalIDE selects the nearest match to the cursor. Then, users can move to the next matches by using the Next vocal command. Their study results show that VocalIDE is very helpful in navigational and select (source code selection) types of commands. Also, participants finished their tasks faster with VocalIDE compare with keyboard or assistive text entry devices.

Although VocalIDE tries to facilitate the code navigation with the coloring ap-

[2]https://www.dropbox.com/s/y1v49lyr0usytw2/DemoVideoFigure.mp4?dl=0

proach, it still suffers from limitations. First, relying entirely on voice modality exposes the system to transcription errors, a common challenge for automatic speech recognition [27]. Second, performing advanced code navigational actions demands multiple micro-interactions in VocalIDE. For instance, if a user wants to check the function body, she has two choices: move the cursor by multiple vocal commands, or say the function name and then move between matches by the Next vocal command. Moreover, VocalIDE compared the system with assistive technologies for people with limited dexterity. Therefore, we are not aware of how VocalIDE would perform for non-disabled users compared to the traditional mouse keyboard setup.



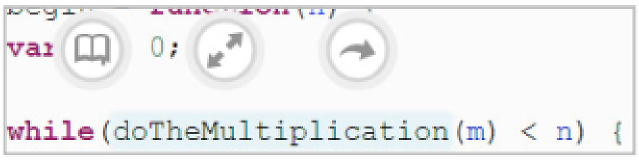
```
public class WeirdDoors {
  public static void main(String[] args) {
    boolean[] doors = new boolean[101];
    for (int a = 1; i <= 100; i++) {
      for (int j = i; j <= 100; j++) {
        if(true) {
          something();
        } else {
          nothing();
        }
      }
    }
  }
}
```

Figure 3: VocalIDE. It gives nine colors to current position surrounding context. Users can utilize colors to move cursor.

2.2. Multimodality

Alternative Multimodality approaches try to use a combination of modalities for replacing mouse-keyboard. There are two types of alternatives Multimodality. The first type adds a keyboard to gaze modality to overcome the limitation in Gaze-based Unimodality systems. The second type replaces the mouse-keyboard with a new Multimodality like voice-gaze.

EyeDE [9] is a prototype which tends to study the effect of using the gaze for source code navigation. EyeDE supports the frequent navigational interactions with source code such as jump to method deceleration, locally expand method body, switch between files. In order to avoid Midas Touch, they use a 2-step confirm button. Users first dwell at the target, then they have to dwell on an icon to confirm the action (Figure 4). The confirm button approach adds extra time-penalty in the result of the second step dwelling. Therefore, EyeDE utilizes a keyboard shortcut as a confirm button instead of a dwelling. Their qualitative evaluation shows that participants, in general, had positive feelings in gaze-based navigation with EyeDE. The quantitative performance and EyeDE performance comparison compare to other modalities are unknown to us since the study did not cover them.

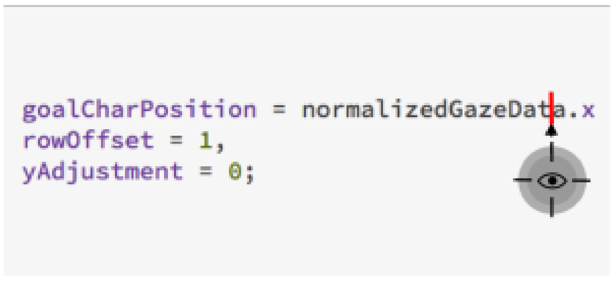


```
var     0;

while(doTheMultiplication(m) < n) {
```

Figure 4: EyeDE. When a user dwells on the method doTheMultiplication, It shows the options to the user. options are (left to right): JavaDoc comment, expand the method body, and go to the method declaration. The user has to again dwell at the desired option and presses a keyboard shortcut to choose it.

EyeNav[25] is a Brackets editor[3] extension that uses a combination of the gaze and keyboard in source code navigation. EyeNav supports both horizontal and vertical scrolling and also code selection and clicking. Users have to use keyboard shortcuts for performing any action in EyeNav (Figure 5). EyeNav considers the center of a user gaze circle as the target point. For example, when a user is gazing at a part of source code, by pressing Alt+Q, EyeNav moves the cursor to the center of the user gaze circle. It is not possible to mention system performance limitations since the study does not have any qualitative or quantitative evaluation. However, one possible limitation would be the high number of keyboard shortcuts that users must memorize to use EyeNav.

[3]http://brackets.io/



```
goalCharPosition = normalizedGazeData.x
rowOffset = 1,
yAdjustment = 0;
```

Figure 5: EyeNav. It considers the center of the user gaze area as the target element. If the x-coordination of center contains no code, it moves the cursor to the nearest element.

Voiceye [22] is a multimodal code editor for HTML/CSS coding where programmers perform code-related activities with Gaze, Voice, and two mechanical switches. The system supports Code Entry, Navigation, Selection, Deletion, and Add Comment. The study focus is facilitating Code Entry with Gaze-Voice Multimodality approach. Go to a line, Up/down (cursor), Left/Right (cursor), and End of Line are the supported navigational actions in Voiceye. Participants use Voice for performing commands such as Go to Line X and use Gaze for code entry via a virtual keyboard. The system uses a mechanical switch for Gaze-click to overcome the Midas Touch issue in gaze-based systems. Besides, participants use another mechanical switch to activate speech input. The study had two experiment steps. In the first one, they test the system with non-disabled programmers to evaluate the system usability. The system received an "OK" system usability score (68.1) and positive feedback from participants. In the second step, they evaluate the system with disabled programmers. Voiceye received a "good and usable" system usability score (74.0). The participants' interview results also show positive opinions about Voiceye.

Although Voiceye obtains good results and feedback, it has some limitations. First, Voiceye performance compared to other modalities (such as traditional mouse-keyboard) for non-disabled programmers is unknown since the study does not cover it. Second, some participants stated that they like to see more navigational actions in the future, which Voiceye does not currently support. Besides, using a mechanical switch for activating speech input adds extra steps for interaction compare to the continuous listening approach.

Study	Modality	Features	Limitation
EyeDE	Gaze + Keyboard	• jump into method body from method call. • highlight all occurrence of a variable/method. • locally expand method body. • documentation lookup. • expand / minimize project navigation.	• time-penalty because of dwelling. • does not support line navigation. • does not support scrolling.
ActiGaze	Gaze	• simulates mouse click with the gaze. • use a coloring approach instead of two-step • confirm button to reduce dwelling time-penalty.	• even with the coloring approach. it still has time-penalty because of dwelling.
CodeGazer	Gaze	• go to definition. find all usages. • list all files. list all methods. • go to a usage. go to a file. • go to a method. go back. go forward.	• dwelling time-penalty. • unintentional scrolling. • User mistakes because of Midas Touch. • does not support direct line navigation. • since all identical variables get same color, it is not possible to choose one specific variable directly.
EyeNav	Gaze + Keyboard	• clicking, scrolling. • single character movement. • code selection.	• frequent keyboard usage.
VocalIDE	Voice	• text entry. selection. deletion. • navigation. replacement. undo. • smart code snippet entry. cut/paste.	• vocal command transcribe problem especially for users with an accent or stuttering issue. • navigation could improve.
Voiceye	Voice + Gaze + mechanical switches	• Code Entry, Navigation, Selection, Deletion, Add Comment • Navigating via Go to line, Up/down (cursor), Left/Right (cursor), and End of Line, • Command by Voice • Coding via Gaze and virtual keyboard • Mechanical switches for speech activation and gaze-click	• Unknown performance compare to other modalities for non-disabled programmers. • It can have more navigational commands in future. • extra steps for interaction because of mechanical switch for speech activation.

Table 1: Summary of the available approaches (uni and multi modal) with listed features and limitations.

3. Research Problem

Alternative Multimodality for code navigation empowers disabled programmers [22]. However, their performance compares to the traditional approach for non-disabled programmers are not well studied. Besides, the scaleability of alternative approaches for having more complex navigational action is not clear. Moreover, there is no study on online navigational interaction in the context of programming. Additionally, Gaze-based approaches suffer from the Midas Touch issue, which decreases accuracy in code navigation. Voice-based approaches also decrease accuracy as a result of transcription error. Furthermore, using Voice for micro cursor moving might not be as efficient as traditional Mouse.

In this Master Thesis, we investigate the feasibility and challenges of utilizing alternative Multimodality approaches for code navigation. Moreover, we try to overcome some of the limitations of existing approaches. Consequently, we aim to investigate the following research questions:

- **RQ1:** How can we naturally integrate another additional modality to the traditional approach for code navigation?
- **RQ2:** How does gaze and voice integration for code navigation compare with the traditional mouse-keyboard approach in terms of time, accuracy, and usability?

4. Methodology

We mention the issues with the mouse-keyboard approach in offline and online navigation before we describe our approach.

Offline navigation: Finding the target element is limited to scrolling, keyboard shortcuts, or using the search functionality in an IDE, which adds navigation overhead to programming [12, 29]. Therefore, it is troublesome to perform more complex searches. For performing the mentioned navigational actions, we have to either define complex keyboard shortcuts or add buttons for each specific action. It is important to note that we are not saying it is impossible to perform these actions with a mouse-keyboard. Instead, our motivation is to investigate if additional modalities lead to improve interactions for offline navigation.

Online navigation: Traditionally, this process has these steps:

1. Move to the search engine
2. Type the desired query
3. Push the search button
4. choose the desired result
5. move back to the last location before the search

Our goal here is to find out a proper way to facilitate this process in terms of interaction using an additional modality.

4.1. Navigational Actions

To investigate the research question, we use a coding environment (Jupyter-notebook), where participants perform both offline and online navigation. Below is the list of actions that our participants perform under different setups. We derived some of these actions from related studies such as CodeGazer and EyeDE. Besides, We design the others based on literature, which indicates the importance of variable tracing in code understanding [7]. Also, we added the relevant navigational commands for an online search. Later in this study, we establish a survey (section 5.1.1) to assess our suggested navigational commands' importance and frequency from the programmers' viewpoint.

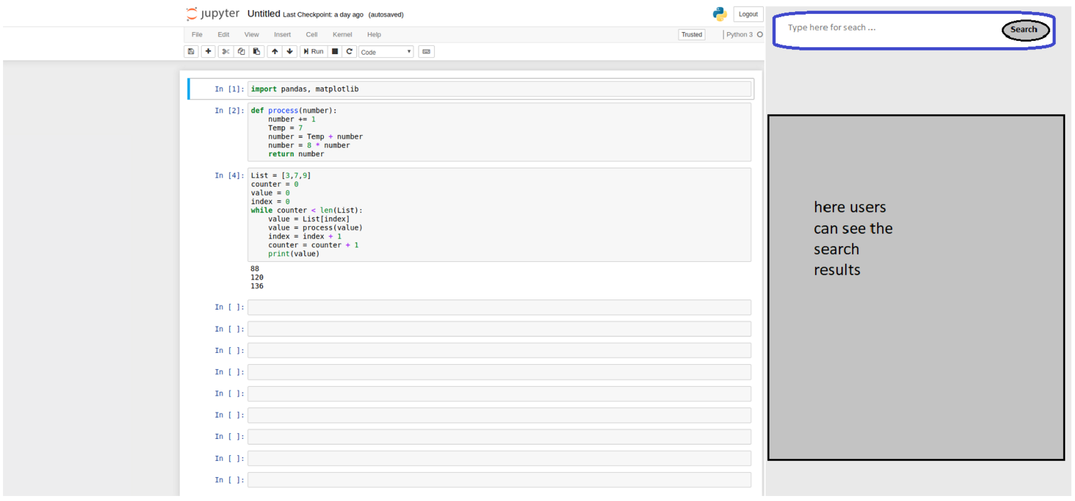
- *Go-To function body*: A user jumps into a function body from a function call [9, 29].
- *Go-Back*: A user jumps back from the navigation destination to the start [29].
- *Move to a variable last modified*: A user checks the last modification of a variable.

- *Move to a variable declaration*: A user checks a variable initialization [29].
- *Move to a variable modifier function*: A user checks the function(s), which modified the variable value.
- *Search Online:*: A user performs a search to find a code snippet or API documentation.
- *Next search result:* A user move to the next search result.

4.2. Implementation

We utilize Jupyter-Notebook[4] as our coding environment. The Jupyter-Notebook is an open-source coding environment that was developed for scientific computing purposes [1]. We choose Jupyter Notebook because of its extension [2] feature. This feature allows us to modify the Jupyter user interface to add our custom features. We use WebKitSpeechRecognition [3] as the speech recognizer engine for speech to text transcription. In order to mitigate the online navigation environment, we add a search bar to the Jupyter notebook. Besides, we show the search result inside the Jupyter notebook by splitting the screen into two parts (Figure 6). This way, we can monitor all user's interactions inside Jupyter notebook. We utilized Stackoverflow API[5] to get search results based on the user query. Moreover, we use Python pydoc[6] for API documentation. We used a MyGaze[7] eye tracker for the Gaze-keyboard setup. We performed the experiment on google Chrome browser[8] and Windows 7 as the operating system.

[4]https://jupyter.org/
[5]https://api.stackexchange.com/docs/advanced-search
[6]https://docs.python.org/3/library/pydoc.html
[7]http://www.mygaze.com/products/mygaze-eye-tracker/
[8]https://www.google.com/chrome/

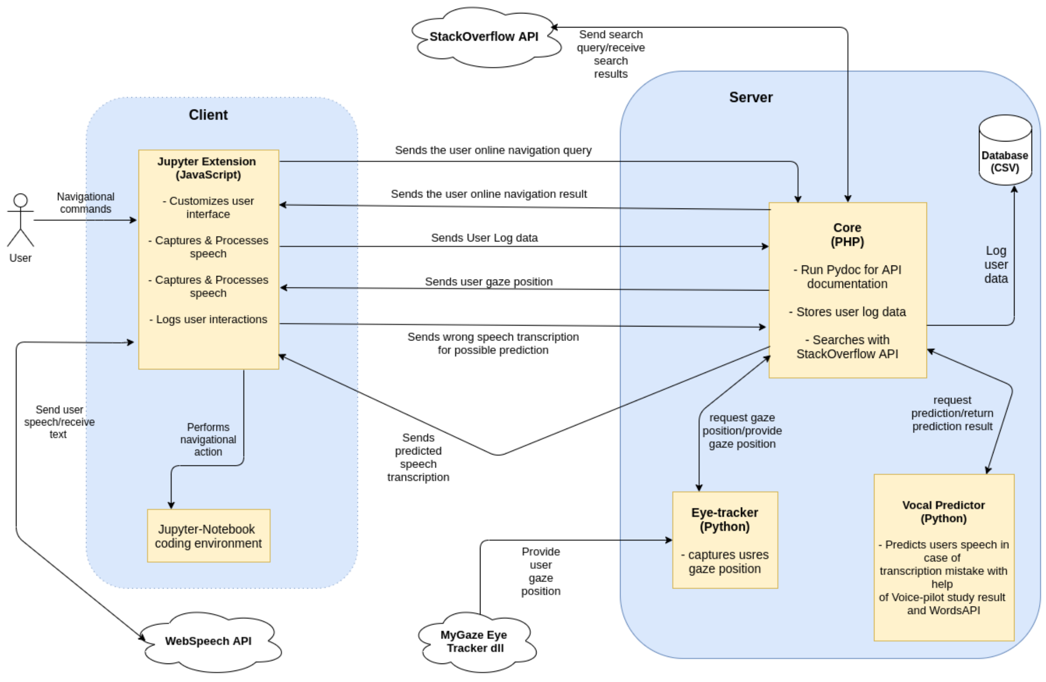


```
In [1]: import pandas, matplotlib

In [2]: def process(number):
            number += 1
            Temp = 7
            number = Temp + number
            number = 8 * number
            return number

In [4]: List = [3,7,9]
        counter = 0
        value = 0
        index = 0
        while counter < len(List):
            value = List[index]
            value = process(value)
            index = index + 1
            counter = counter + 1
            print(value)

        88
        120
        136

In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
```

Figure 6: We add the search part to the right side of the Jupyter Notebook.

4.2.1. Architecture

The system architecture is client-server (Figure 7). The client is a Jupyter-notebook extension, which is written in Javascript. We developed three Jupyter Notebook extensions for the three setups (sections 4.3, 4.4, 4.5) we have in this study. The client is responsible for all user-related activities. First, it modifies the notebook interface to add our additional features like the search bar. Second, it captures vocal commands and keyboard shortcuts and performs the navigational actions. Finally, it logs all user interactions and sends them to the server. The server stores the users' interaction log. It returns search results and API documentation to the client based on the user input. Moreover, it captures users' gaze positions and returns them to the client. Further, it runs a prediction system (section 4.5.1) on users' transcript speech in case of speech-to-text transcription error.



StackOverflow API
Send search query/receive search results
Client
Server
Database (CSV)
Jupyter Extension (JavaScript)
- Customizes user interface
- Captures & Processes speech
- Captures & Processes speech
- Logs user interactions
User
Navigational commands
Sends the user online navigation query
Sends the user online navigation result
Sends User Log data
Sends user gaze position
Sends wrong speech transcription for possible prediction
Core (PHP)
- Run Pydoc for API documentation
- Stores user log data
- Searches with StackOverflow API
Log user data
Send user speech/receive text
Performs navigational action
Sends predicted speech transcription
request gaze position/provide gaze position
request prediction/return prediction result
Jupyter-Notebook coding environment
Eye-tracker (Python)
- captures usres gaze position
Vocal Predictor (Python)
- Predicts users speech in case of transcription mistake with help of Voice-pilot study result and WordsAPI
Provide user gaze position
WebSpeech API
MyGaze Eye Tracker dll

Figure 7: System architecture

4.2.2. Setups

We have three setups in this study.

- **Mouse-Keyboard**: This setup is designed based on the traditional mouse-keyboard approach, which is the natural way of code navigation for programmers. In this setup, we use keyboard shortcuts for navigational actions and mouse for moving cursor and scrolling. (section 4.3)
- **Gaze-Keyboard**: In this setup, we use gaze as an additional modality for the traditional keyboard approach. We use gaze for navigational actions, moving the cursor, and scrolling. We utilize a keyboard for typing and the mechanical switch for gaze click (section 4.4).
- **Voice-Mouse**: In the third setup, we perform navigational actions by adding voice as an additional modality to the mouse. We utilize voice for the navigational commands and mouse for moving the cursor and scrolling (section 4.5).

4.2.3. Workflow

Figure 8 demonstrates our experiment workflow. We have two familiarization steps. In the first one, we perform a test experiment for our participants, and in the second, we let our participants perform a test experiment. To neutralize the effect of setup order on participants' experience, we divide them into six groups. Then, we perform the experimental setups with different orders for each group (Table 2).

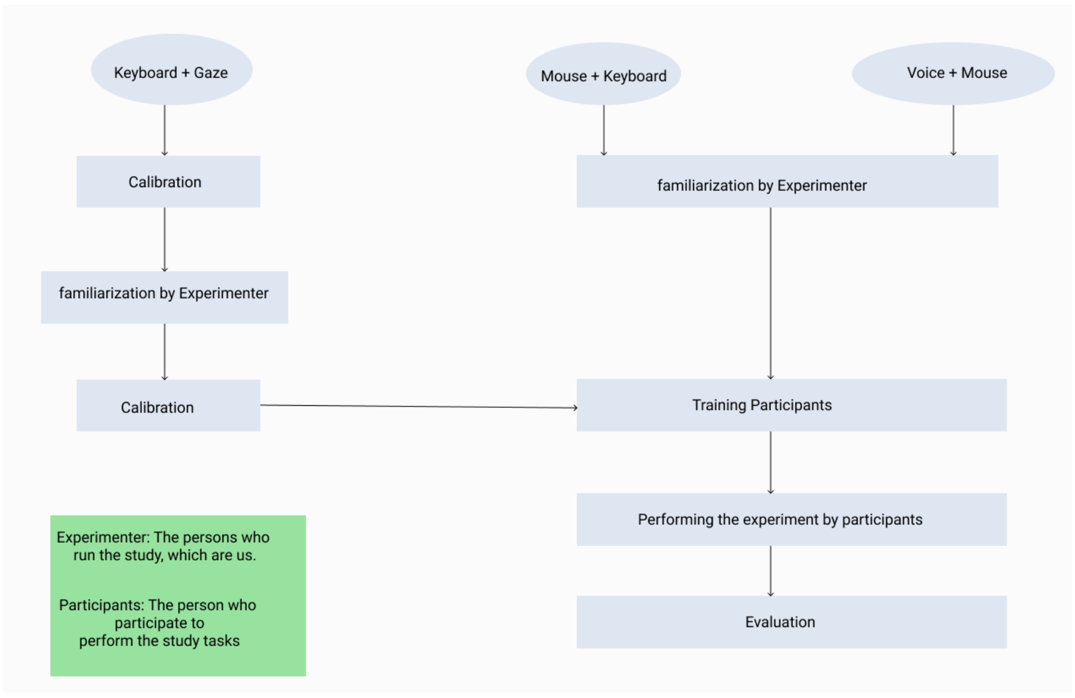

Keyboard + Gaze
Mouse + Keyboard
Voice + Mouse
Calibration
familiarization by Experimenter
familiarization by Experimenter
Calibration
Training Participants
Performing the experiment by participants
Evaluation
Experimenter: The persons who run the study, which are us.
Participants: The person who participate to perform the study tasks

Figure 8: Study workflow.

Participant Groups			
Group	**Gaze + Keyboard**	**Mouse + Keyboard**	**Voice + Mouse**
A	1	2	3
B	1	3	2
C	2	1	3
D	2	3	1
E	3	1	2
F	3	2	1

Table 2: Participant Groups table. The numbers indicate the order of a setup for a group. For instance, For group B the order is: 1- Gaze + Keyboard, 2- Voice + Mouse, 3- Mouse + Keyboard.

4.3. Setup: Mouse + Keyboard

In this setup, participants use mouse and keyboard for code navigation, focusing the keyboard shortcuts. We assign a keyboard shortcut to each navigational activity (Table 3). We chose the same approach as the standard "Find" (Ctrl+f) action in the Chrome browser and Jupyter Notebook. For instance, for moving to a variable declaration, a participant first use the associated keyboard shortcut. Then, he/she types the target identifier's name in the modal[9] and presses Enter (Figure 10). Participants can perform online navigation by typing a question in the search bar to search for related code snippets (Figure 9). Participants can switch between online search results by using the keyboard.

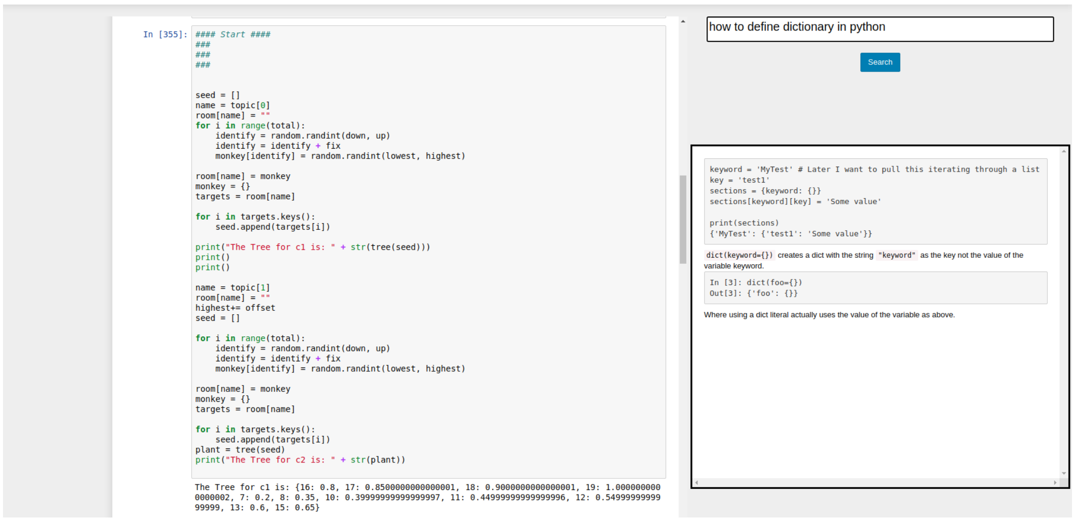
4.3.1. Challenges

The main challenge for us was choosing the proper keyboard shortcuts for each navigational action in this setup. Since Jupyter Notebook is a browser-based editor, we had three groups of shortcuts (Jupyter default shortcuts, Web browser shortcuts, and Operating system shortcuts) that limited our options. We had to disable Jupyter Notebook default shortcuts to avoid possible collision between them and our setup shortcuts. However, it was cumbersome or impossible to manipulate the Web browser (Chrome) and Operating System (Windows 7) shortcuts. Initially, we did not suggest the Ctrl key as part of our shortcuts and only had the Alt key in them. However, we noticed that many Alt-only shortcuts are reserved (for example, Alt+d selects the address bar in Chrome). Therefore, we added the Ctrl key to our shortcuts (Table 3). Also, we ran into a problem with the Search shortcut (Ctrl + Alt+ h). At first, we considered Ctrl+Alt+r for this navigational action. However, during the test experiment (before the actual experiment), we noticed sometimes the participant press Ctrl+r mistakenly, which reloads the entire page. Since by reloading the page, we need to repeat the task, we considered to change the shortcut to Ctrl+Alt+h.

List of keyboard shortcuts	
Command	**Description**
Ctrl + Alt + g	- Go-To function body
Ctrl + b	- Go-Back
Ctrl + Alt + k	- Move to a variable last modified
Ctrl + Alt + d	- Move to a variable declaration
Ctrl + Alt + n	- Move to a variable modifier function
Ctrl + Alt + p	- Trigger the search for API documentation
Ctrl + Alt + h	- Move the cursor to the search box
Ctrl + q	- Move to the next search result

Table 3: Keyboard shortcuts and their associated navigational actions. Participants had this listed printed in front of them during the experiment.

[9] https://getbootstrap.com/docs/4.0/components/modal/



```
In [355]: #### Start ####
          ###
          ###
          ###


          seed = []
          name = topic[0]
          room[name] = ""
          for i in range(total):
              identify = random.randint(down, up)
              identify = identify + fix
              monkey[identify] = random.randint(lowest, highest)

          room[name] = monkey
          monkey = {}
          targets = room[name]

          for i in targets.keys():
              seed.append(targets[i])

          print("The Tree for c1 is: " + str(tree(seed)))
          print()
          print()

          name = topic[1]
          room[name] = ""
          highest+= offset
          seed = []

          for i in range(total):
              identify = random.randint(down, up)
              identify = identify + fix
              monkey[identify] = random.randint(lowest, highest)

          room[name] = monkey
          monkey = {}
          targets = room[name]

          for i in targets.keys():
              seed.append(targets[i])
          plant = tree(seed)
          print("The Tree for c2 is: " + str(plant))

          The Tree for c1 is: {16: 0.8, 17: 0.8500000000000001, 18: 0.9000000000000001, 19: 1.000000000
          0000002, 7: 0.2, 8: 0.35, 10: 0.39999999999999997, 11: 0.44999999999999996, 12: 0.54999999999
          99999, 13: 0.6, 15: 0.65}
```

how to define dictionary in python

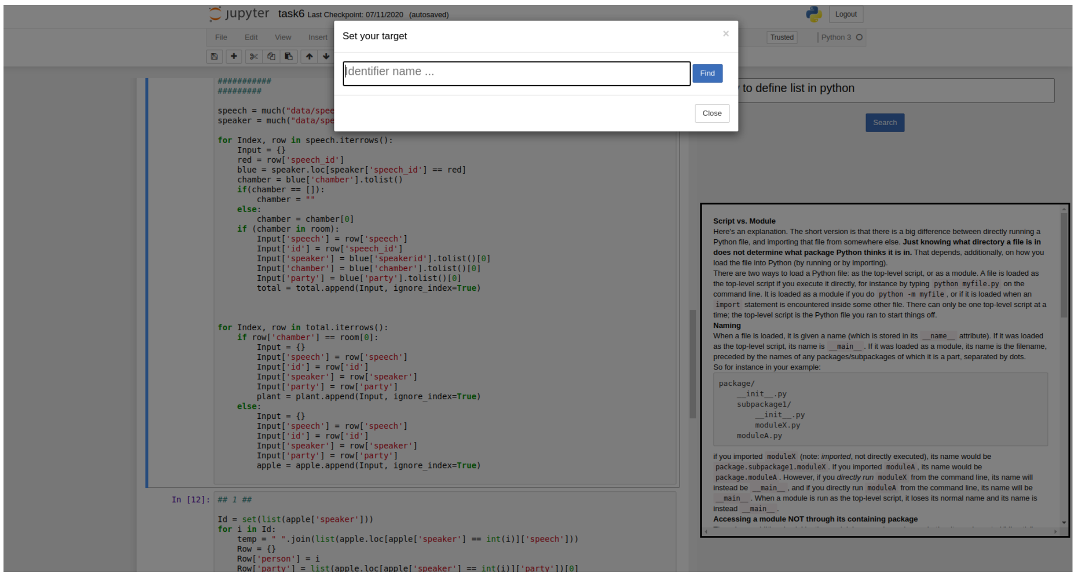
```
keyword = 'MyTest' # Later I want to pull this iterating through a list
key = 'test1'
sections = {keyword: {}}
sections[keyword][key] = 'Some value'

print(sections)
{'MyTest': {'test1': 'Some value'}}
```

`dict(keyword={})` creates a dict with the string `"keyword"` as the key not the value of the variable keyword.

```
In [3]: dict(foo={})
Out[3]: {'foo': {}}
```

Where using a dict literal actually uses the value of the variable as above.

Figure 9: Mouse + Keyboard. Participants can use the Ctrl+Alt+h to switch to the search box and type down the search query. After pressing the Enter key or clicking the search button, the search result will show up on the right side.



```
          ###########
          #########

          speech = much("data/spee
          speaker = much("data/spe

          for Index, row in speech.iterrows():
              Input = {}
              red = row['speech_id']
              blue = speaker.loc[speaker['speech_id'] == red]
              chamber = blue['chamber'].tolist()
              if(chamber == []):
                  chamber = ""
              else:
                  chamber = chamber[0]
              if (chamber in room):
                  Input['speech'] = row['speech']
                  Input['id'] = row['speech_id']
                  Input['speaker'] = blue['speakerid'].tolist()[0]
                  Input['chamber'] = blue['chamber'].tolist()[0]
                  Input['party'] = blue['party'].tolist()[0]
                  total = total.append(Input, ignore_index=True)



          for Index, row in total.iterrows():
              if row['chamber'] == room[0]:
                  Input = {}
                  Input['speech'] = row['speech']
                  Input['id'] = row['id']
                  Input['speaker'] = row['speaker']
                  Input['party'] = row['party']
                  plant = plant.append(Input, ignore_index=True)
              else:
                  Input = {}
                  Input['speech'] = row['speech']
                  Input['id'] = row['id']
                  Input['speaker'] = row['speaker']
                  Input['party'] = row['party']
                  apple = apple.append(Input, ignore_index=True)

In [12]:  ## 1 ##

          Id = set(list(apple['speaker']))
          for i in Id:
              temp = " ".join(list(apple.loc[apple['speaker'] == int(i)]['speech']))
              Row = {}
              Row['person'] = i
              Row['party'] = list(apple.loc[apple['speaker'] == int(i)]['party'])[0]
```

to define list in python

Script vs. Module
Here's an explanation. The short version is that there is a big difference between directly running a Python file, and importing that file from somewhere else. **Just knowing what directory a file is in does not determine what package Python thinks it is in.** That depends, additionally, on how you load the file into Python (by running or by importing).
There are two ways to load a Python file: as the top-level script, or as a module. A file is loaded as the top-level script if you execute it directly, for instance by typing `python myfile.py` on the command line. It is loaded as a module if you do `python -m myfile`, or if it is loaded when an `import` statement is encountered inside some other file. There can only be one top-level script at a time; the top-level script is the Python file you ran to start things off.
Naming
When a file is loaded, it is given a name (which is stored in its `__name__` attribute). If it was loaded as the top-level script, its name is `__main__`. If it was loaded as a module, its name is the filename, preceded by the names of any packages/subpackages of which it is a part, separated by dots.
So for instance in your example:

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
    moduleA.py
```

if you imported `moduleX` (note: *imported*, not directly executed), its name would be `package.subpackage1.moduleX`. If you imported `moduleA`, its name would be `package.moduleA`. However, if you *directly run* `moduleX` from the command line, its name will instead be `__main__`, and if you directly run `moduleA` from the command line, its name will be `__main__`. When a module is run as the top-level script, it loses its normal name and its name is instead `__main__`.
Accessing a module NOT through its containing package

Figure 10: Mouse + Keyboard. After the participant presses a navigational shortcut, this modal pops up where he/she can type the target identifier's name.

4.4. Setup: Keyboard + Gaze

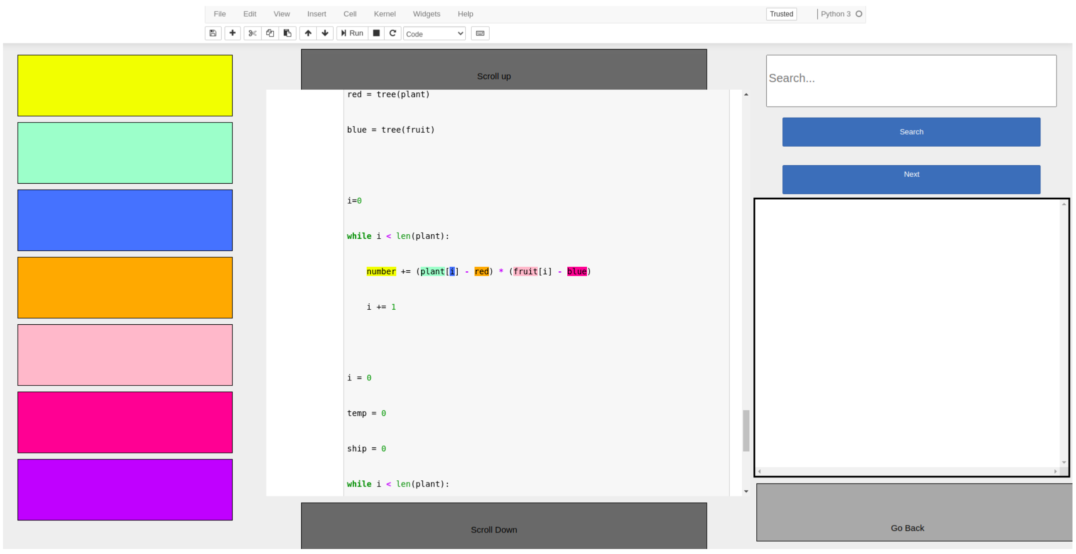
We plan to utilize gaze as an additional modality for performing offline and online navigation in this setup. Related studies such as CodeGazer and EyeDE try to replace the mouse with the gaze because of their similar code navigation behavior.

Therefore, we also replace the mouse with the Gaze in this setup. CodeGazer introduces a novel coloring approach for selecting an identifier in a code. Besides, CodeGazer tries to minimize the dwelling time-penalty by utilizing a low dwell time (300ms). However, their study result shows that the low dwell time might be negatively related to accuracy. Therefore, we take advantage of the Voiceye [22] approach to overcome the dwelling issue. Plus, CodeGazer does not support line-wise navigation in the code that means users cannot directly navigate to a specific line inside the code (Figure 2). We implemented our gaze setup based on line-navigation (Figure 11). When users gaze at a line in our setup, the system gives colors to the identifiers inside the target line. The line coloring is based on unique identifiers, which means if an identifier has multiple occurrences in the line, we color only one of them. This way, we can have a longer code line since our colors are limited. Like CodeGazer, we give seven colors to the identifiers inside the target line. A user can then pick a target for navigation by gazing at the side colors (Figure 11).

On the other hand, for overcoming the dwelling issue, we use the Voiceye approach. We highlight the target color button inside the user gaze area (by using an eye symbol, Figure 12). Then, the user can select a color by pressing the Ctrl key on the keyboard. EyeDE and EyeNav use the Alt key for performing actions. However, locating the Alt key on the keyboard while gazing at the code might be difficult for users. Voiceye used a mechanical switch to perform actions because they did not have the keyboard modality. Since We have the keyboard modality, we decided to use the Ctrl key to avoid using an additional modality. After a user selects a target identifier, we show her/him a modal which contains navigational shortcuts as buttons (Figure 12). A user can gaze at the target button and press the Ctrl key. Users can gaze at the search bar to move the cursor there and then type their query in the search bar for entering a search query. After typing the search query, users gaze at the Search button and press the Ctrl key. Users can move to the next result by gazing at the Next button and pressing the Ctrl button. For Go-Back navigational action, users can gaze at the Go-back button (right side at the bottom) and use the Ctrl key. For scrolling up/down, users gaze at the associated scroll button on the top/bottom part of the screen and use the Ctrl key to start the scrolling. Users have to do the same procedure for stopping the scrolling.

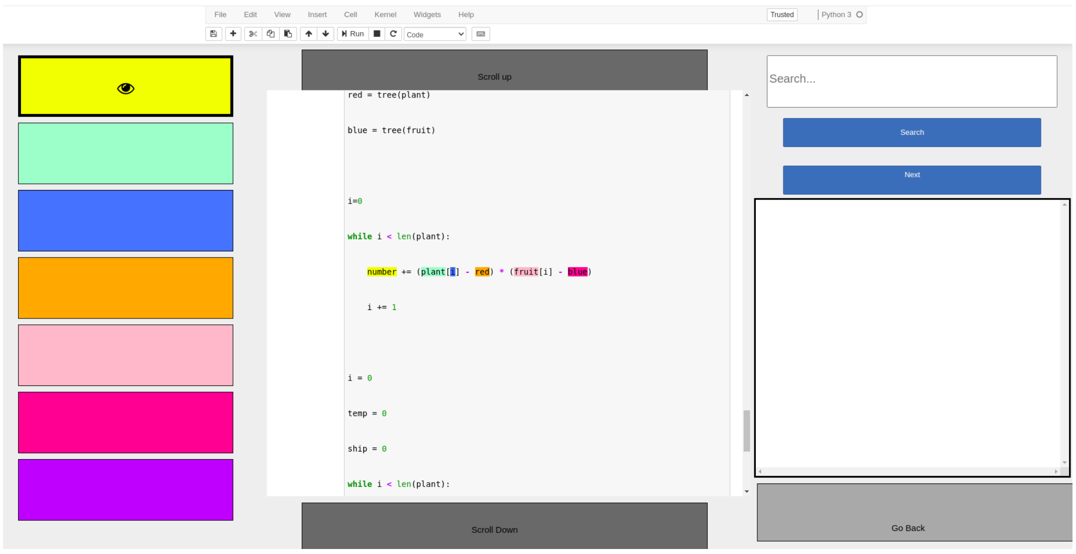
4.4.1. Challenges

The first challenge we faced was with the line-wise gaze selection. The issue was that code lines were too close to each other, which was not feasible to select with the eye tracker gaze position. To overcome this challenge, we added extra margins to each line's top and bottom (Figure 11). The second challenge was with our scrolling approach. We added an action key (Ctrl key) to overcome unintentional scrolling. However, this approach raised another issue. Gazing at the scroll buttons and press the Ctrl key adds a time delay in the scrolling, which leads to missing the target scrolling point because of scrolling speed. To overcome this issue, we slowed down the scrolling for gaze setup to give enough time to the user to stop the scrolling action on the desired place. We are aware that this adds time-penalties to our Gaze setup. However, we could overcome unintentional scrolling with our approach.



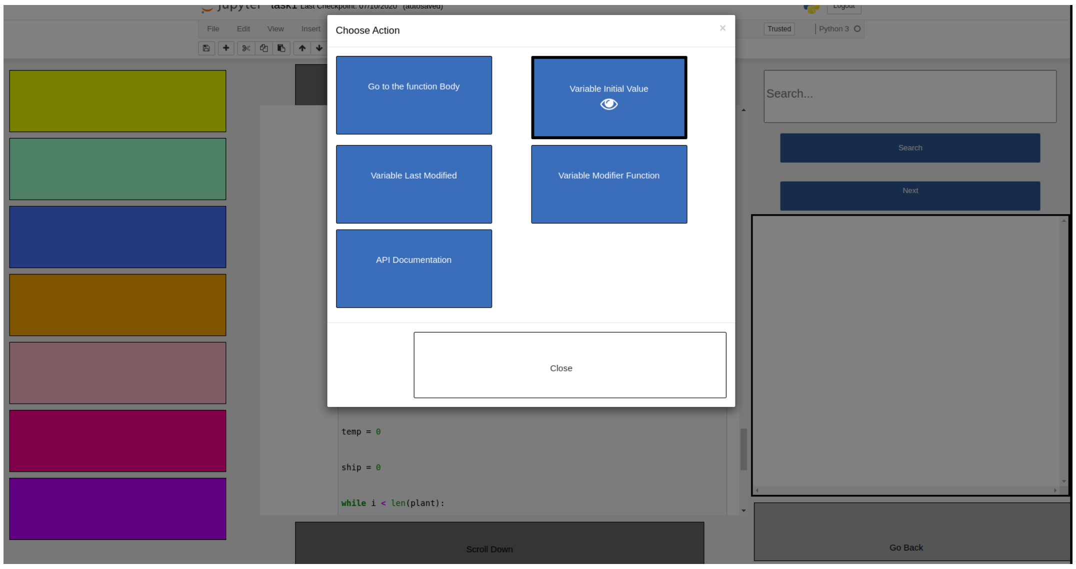
```
red = tree(plant)

blue = tree(fruit)


i=0

while i < len(plant):

    number += (plant[i] - red) * (fruit[i] - blue)

    i += 1


i = 0

temp = 0

ship = 0

while i < len(plant):
```

Figure 11: Gaze + Keyboard. Users gaze at the line which contains the target identifiers. We added extra top/bottom margin to each line to make line-wise selection feasible.



```
red = tree(plant)

blue = tree(fruit)


i=0

while i < len(plant):

    number += (plant[i] - red) * (fruit[i] - blue)

    i += 1


i = 0

temp = 0

ship = 0

while i < len(plant):
```

Figure 12: Gaze + Keyboard. An eye symbol appears on the color box which the user is gazing at. Then the user presses the Ctrl key to select the desire color (yellow in this example)



```
temp = 0

ship = 0

while i < len(plant):
```

Figure 13: Gaze + Keyboard. After choosing the target color and pressing the Ctrl key, a modal view pops up. Then the user can select the desire action by gazing at it (eye symbol) and pressing the Ctrl key.

4.5. Setup: Voice + Mouse

In this setup, participants perform offline and online navigation with the voice (Figure 14). Participants can use a mouse for simple navigational interaction, such as scrolling. We assign vocal commands based on the survey results (section 6.1) to each navigational action of this study (Figure 16). Voiceye uses a mechanical switch to activate speech input, which adds an extra interaction step. Our system is based on continuous listening, which means our speech-to-text engine is continuously ready to accept vocal commands without using a trigger button. For instance, for Move to a Variable (names X) last modified, a participant can say "last X." For Online navigation, participants can use the "documentation" and "Search" vocal commands. For instance, "Search how to define dataframe in python" triggers the search functionality with the query "how to define dataframe in python" (Figure 15). Participants can switch between search results by using the "Next" vocal command.

4.5.1. Prediction system

We implemented a prediction system to reduce the number of transcription error by speech-to-text API. We utilize the result of the voice pilot study (section 5.1.2). When a transcription mistake happens (wrong command, wrong target name, or both), the system sends a prediction request to the server. The server compares the tokens (separated by space) inside the speech with our database. The database is a

Python dictionary[10] in which keys are the commands/identifiers in our tasks (section 5.5), and values are the words with similar rhymes. In case we cannot predict the command, we look at the target identifier to predict the command based on that. For example, if the target identifier is an API, we can say the command is "documentation" most probably. We separated our prediction into two steps. The first step is the prediction based on our voice pilot study result, and the second step is based on the WordsAPI (section 5.1.2) results. This way, we can observe each step's performance separately in our final evaluation.

4.5.2. Challenges

Originally, we named this setup "Voice + Keyboard + Mouse." The reason was that we planned to use the Alt key to trigger the speech API listening. Voiceye also practiced this approach by using a mechanical switch. However, we noticed some issues with this approach. After the voice pilot study (section 5.1.2), some participants stated that they could not finish the command after clicking the record button. The reason was related to the speech API time limitation for listening. In order to avoid this effect, we changed our approach to continuous listening. We can also reduce the extra interaction step, which was pressing the Alt key on each command. Therefore, we changed the setup name to "Voice + Mouse" since there is no keyboard usage in the new version. The next challenge was with the API names. Many of the Python APIs have the character "." which increase the chance of transcription mistake and make them difficult to pronounce. To overcome this, we mapped the API functions to their library. For example, for the API "divide" in numpy.divide[11], participants could say "documentation divide" instead of "documentation numpy.divide". Beyond, we utilized the "import as"[12] feature in Python to rename some APIs to a shorter name. We did the same for other setups (Gaze+Keyboard, Mouse+Keyboard) to make the situation identical for all. For instance, in the Mouse+Keyboard setup, participants only type down the "divide" and not the entire "numpy.divide". In the end, we had to add the "You said" box (Figure 15) to our setup, where participants can see their transcription outcome. This was recommended by Voiceye, where participants complained about lacking of this feature.

[10]`https://docs.python.org/3/tutorial/datastructures.html`
[11]`https://docs.scipy.org/doc/numpy/reference/generated/numpy.divide.html`
[12]`https://docs.python.org/2.0/ref/import.html`

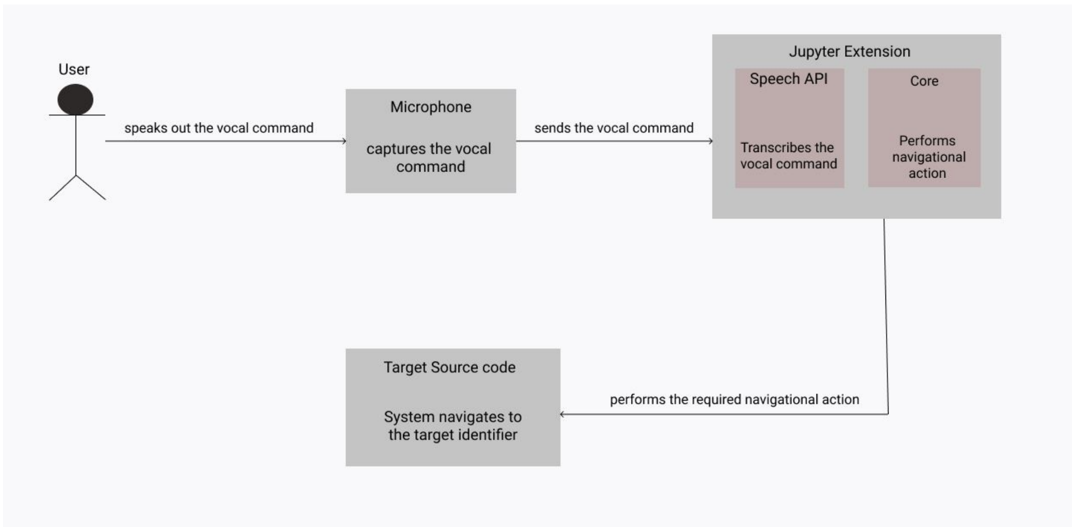
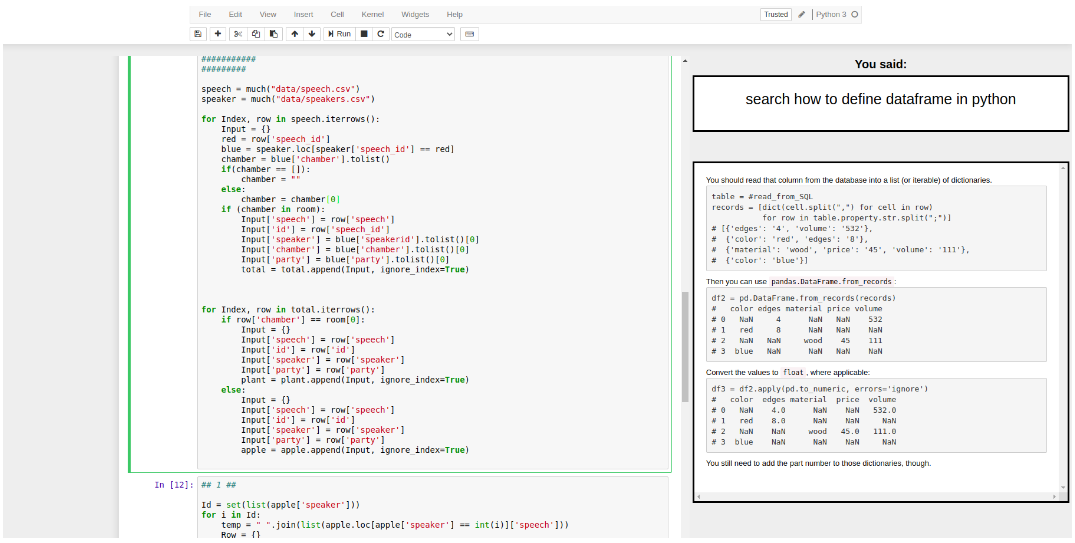

User
speaks out the vocal command
Microphone
captures the vocal command
sends the vocal command
Jupyter Extension
Speech API
Transcribes the vocal command
Core
Performs navigational action
performs the required navigational action
Target Source code
System navigates to the target identifier

Figure 14: Voice + Mouse. Users speak out their desire navigational vocal command. Then, the system transcribes it and performs the navigational action on the source code.



```
###########
#########

speech = much("data/speech.csv")
speaker = much("data/speakers.csv")

for Index, row in speech.iterrows():
    Input = {}
    red = row['speech_id']
    blue = speaker.loc[speaker['speech_id'] == red]
    chamber = blue['chamber'].tolist()
    if(chamber == []):
        chamber = ""
    else:
        chamber = chamber[0]
    if (chamber in room):
        Input['speech'] = row['speech']
        Input['id'] = row['speech_id']
        Input['speaker'] = blue['speakerid'].tolist()[0]
        Input['chamber'] = blue['chamber'].tolist()[0]
        Input['party'] = blue['party'].tolist()[0]
        total = total.append(Input, ignore_index=True)



for Index, row in total.iterrows():
    if row['chamber'] == room[0]:
        Input = {}
        Input['speech'] = row['speech']
        Input['id'] = row['id']
        Input['speaker'] = row['speaker']
        Input['party'] = row['party']
        plant = plant.append(Input, ignore_index=True)
    else:
        Input = {}
        Input['speech'] = row['speech']
        Input['id'] = row['id']
        Input['speaker'] = row['speaker']
        Input['party'] = row['party']
        apple = apple.append(Input, ignore_index=True)
```

```
In [12]: ## 1 ##

Id = set(list(apple['speaker']))
for i in Id:
    temp = " ".join(list(apple.loc[apple['speaker'] == int(i)]['speech']))
    Row = {}
```

You said:

search how to define dataframe in python

You should read that column from the database into a list (or iterable) of dictionaries.

```
table = #read_from_SQL
records = [dict(cell.split(",") for cell in row)
           for row in table.property.str.split(";")]
# [{'edges': '4', 'volume': '532'},
#  {'color': 'red', 'edges': '8'},
#  {'material': 'wood', 'price': '45', 'volume': '111'},
#  {'color': 'blue'}]
```

Then you can use `pandas.DataFrame.from_records`:

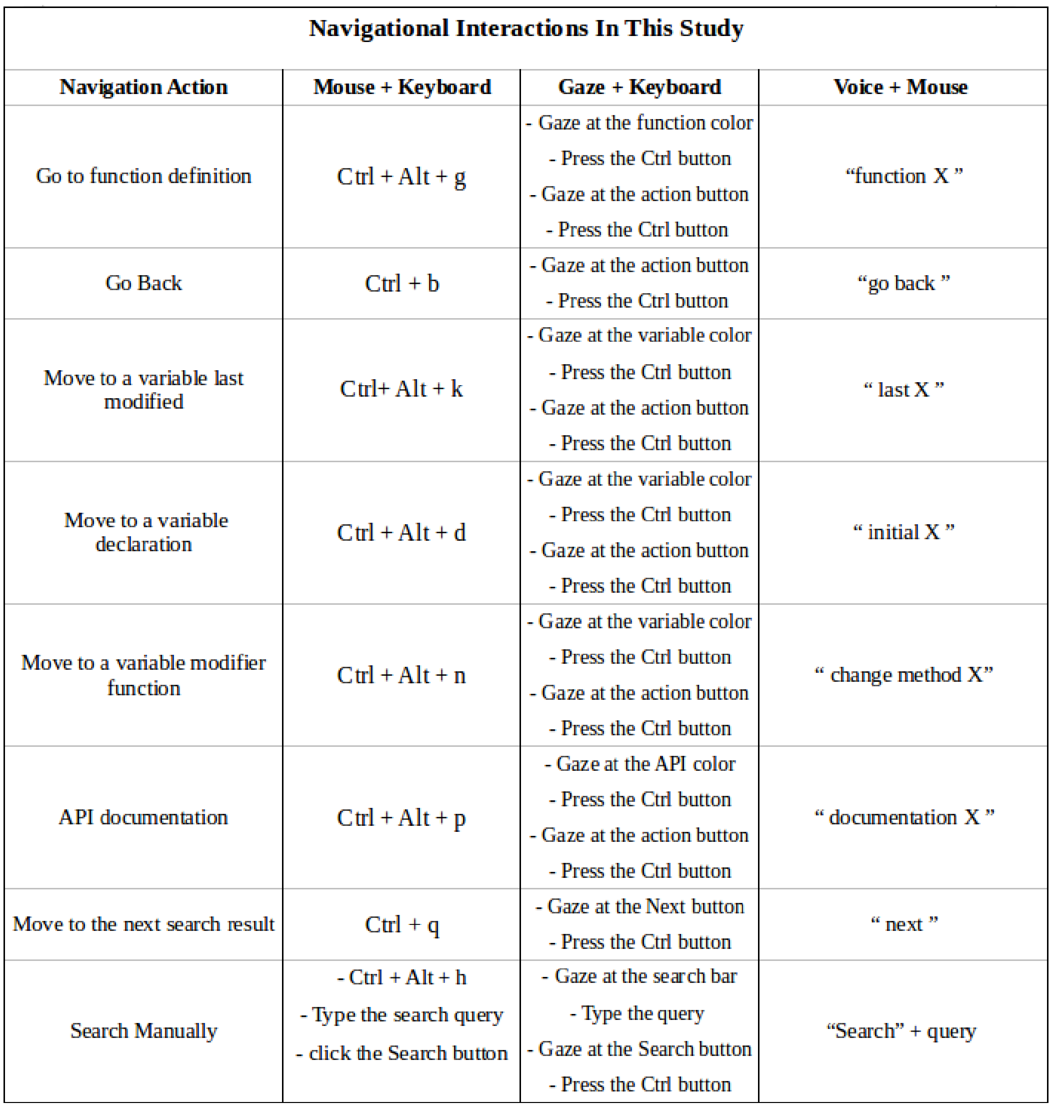
```
df2 = pd.DataFrame.from_records(records)
#   color edges material price volume
# 0   NaN     4      NaN   NaN    532
# 1   red     8      NaN   NaN    NaN
# 2   NaN   NaN     wood    45    111
# 3  blue   NaN      NaN   NaN    NaN
```

Convert the values to `float`, where applicable:

```
df3 = df2.apply(pd.to_numeric, errors='ignore')
#   color  edges material  price  volume
# 0   NaN    4.0      NaN    NaN   532.0
# 1   red    8.0      NaN    NaN     NaN
# 2   NaN    NaN     wood   45.0   111.0
# 3  blue    NaN      NaN    NaN     NaN
```

You still need to add the part number to those dictionaries, though.

Figure 15: Voice + Mouse. Participants could see their speech transcription outcome inside the "You said" box.



Navigational Interactions In This Study			
Navigation Action	**Mouse + Keyboard**	**Gaze + Keyboard**	**Voice + Mouse**
Go to function definition	Ctrl + Alt + g	- Gaze at the function color - Press the Ctrl button - Gaze at the action button - Press the Ctrl button	"function X "
Go Back	Ctrl + b	- Gaze at the action button - Press the Ctrl button	"go back "
Move to a variable last modified	Ctrl+ Alt + k	- Gaze at the variable color - Press the Ctrl button - Gaze at the action button - Press the Ctrl button	" last X "
Move to a variable declaration	Ctrl + Alt + d	- Gaze at the variable color - Press the Ctrl button - Gaze at the action button - Press the Ctrl button	" initial X "
Move to a variable modifier function	Ctrl + Alt + n	- Gaze at the variable color - Press the Ctrl button - Gaze at the action button - Press the Ctrl button	" change method X"
API documentation	Ctrl + Alt + p	- Gaze at the API color - Press the Ctrl button - Gaze at the action button - Press the Ctrl button	" documentation X "
Move to the next search result	Ctrl + q	- Gaze at the Next button - Press the Ctrl button	" next "
Search Manually	- Ctrl + Alt + h - Type the search query - click the Search button	- Gaze at the search bar - Type the query - Gaze at the Search button - Press the Ctrl button	"Search" + query

Figure 16: Navigational actions in this study which users perform in three setups. In the Voice + Mouse setup, the X is the target identifier name. In each setup, we train participants before performing the actual experiment. Also, participant have the printed list of keyboard shortcuts and vocal commands during the experiment.

5. Experiment and Evaluation

The goal of the evaluation is to compare the setups in this study. Each participant participates in two experimental phases (phase1 and phase2). In each phase, participants participate in three experiment sessions. In each session, participants perform three Python tasks (section 5.5) with one of the setups in this study. Figure 17 shows the relation between a participant, phases, sessions, and tasks.

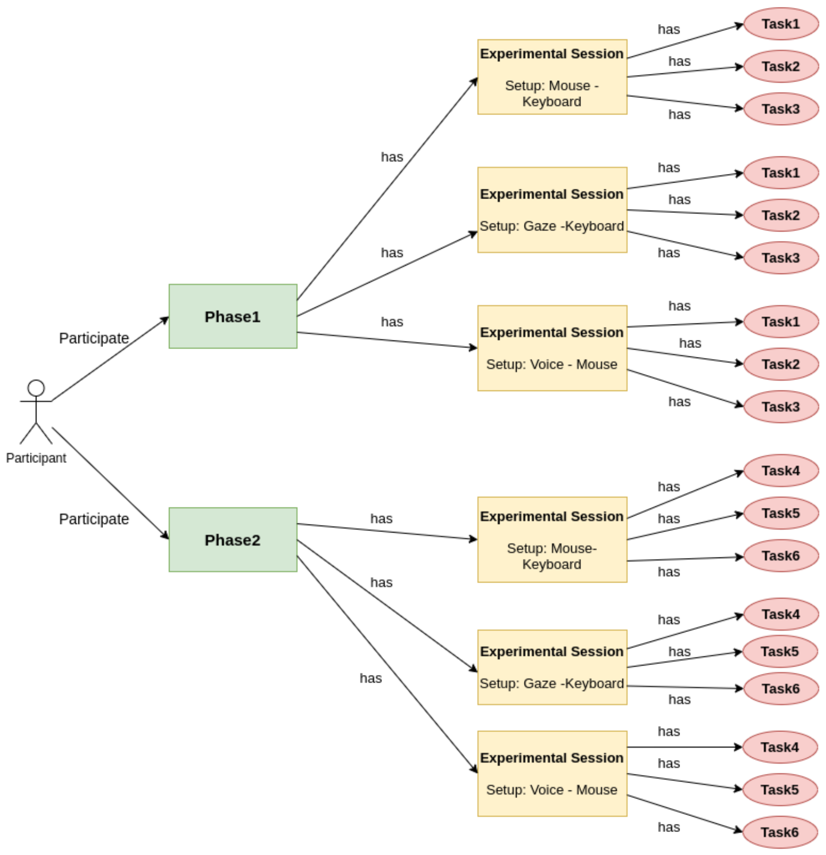
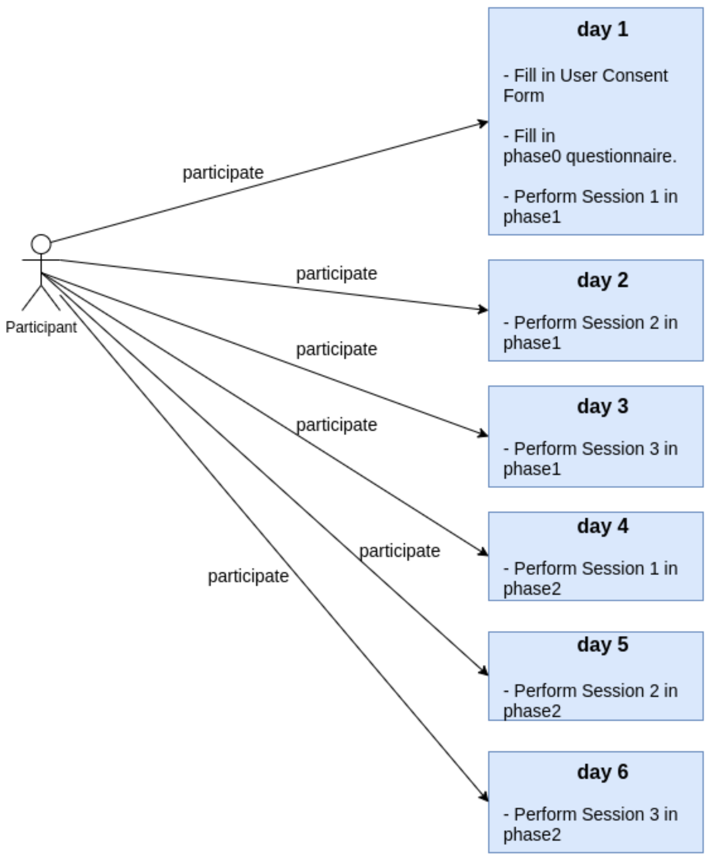


Figure 17: The relations between phases, sessions, and tasks for a participant in this study.

Each participant performs his/her sessions on six different days. The order of performing setups is different for participants (Table 2). In the first session, participants fill in the user consent form and an experiment questionnaire. We call the experiment questionnaire "phase0" in this study, as described in section 5.2. Figure 18 shows the experiment process for one participant as an example.



day 1
- Fill in User Consent Form
- Fill in phase0 questionnaire.
- Perform Session 1 in phase1
day 2
- Perform Session 2 in phase1
day 3
- Perform Session 3 in phase1
day 4
- Perform Session 1 in phase2
day 5
- Perform Session 2 in phase2
day 6
- Perform Session 3 in phase2
participate
Participant

Figure 18: The experiment procedure for a participant in this study.

5.1. Pilot Study

We performed two pilot studies before implementing our main experiment. First, we built a survey to investigate the importance of navigation actions in this study (section 3.2) from the user viewpoint. Second, we performed a pilot study to reduce the impact of transcription errors in our voice-based setup.

5.1.1. Survey

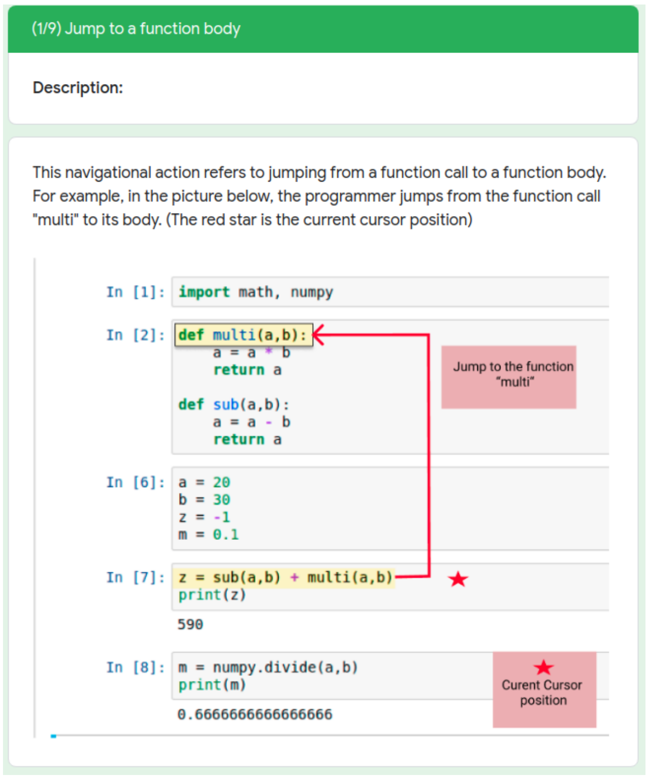
This survey's main goals were to evaluate the importance of navigational actions and design natural vocal commands for them. Table 4 shows the questions that we used in this survey. The survey was implemented in google forms[13].

[13]`https://www.google.com/forms/about/`

At first, we shortly described the code navigation conception. Then, we ask participants to answers questions regarding each navigational action. For each action, we provided a short definition with an example screenshot to explain it for participants (Figure 19). Next, we ask our participants regarding their interest in each navigational action (question 1, question 2). For each navigational action, we also ask the participant suggestions for the vocal command (question 3). At the end of the survey, we ask our participants to suggest their desired navigational actions and their vocal command in case we did not mention them (question 4, question 5). We also ask about participants' level of expertise in programming and Python programming (question 6, question 7, question 8). The last three questions' goal is to investigate the possible relation between participants' programming level and their interest in navigational actions. Appendix A shows the screenshots of the survey.

Question	**Options**
1- How often do you use such navigation in your programming setup?	6-point scale
2- How important do you think is this navigational action (jump from function call to function body?)	6-point scale
3- If you had a voice assistant like Siri or Google, and you want to give a voice command to perform this navigational task, what would that voice command be?	open
4- Which navigational action(s) do you think is/are important which was not mentioned in the survey? In case of multiple answers, please separate them by comma (,)	open
5- If you had a voice assistant like Siri or Google, and you want to give a voice command to perform this navigational task, what would that voice command be?(In case of multiple answers, please separate them by comma)	open
6- How many years experience do you have in programming?	none, less than 1 year, between 1 year and 3 years, between 3 year and 5 years, More than 5 years
7- How many years experience do you have with Python programming language?	none, less than 1 year, between 1 year and 3 years, between 3 year and 5 years, More than 5 years
8- How do you rate your level of expertise in Python programming?	6-point scale

Table 4: The questions in the survey pilot study. The main goals are to assess the navigational actions and design proper vocal commands for them.


(1/9) Jump to a function body

Description:

This navigational action refers to jumping from a function call to a function body. For example, in the picture below, the programmer jumps from the function call "multi" to its body. (The red star is the current cursor position)

```
In [1]: import math, numpy

In [2]: def multi(a,b):
            a = a * b
            return a

        def sub(a,b):
            a = a - b
            return a

In [6]: a = 20
        b = 30
        z = -1
        m = 0.1

In [7]: z = sub(a,b) + multi(a,b)
        print(z)

        590

In [8]: m = numpy.divide(a,b)
        print(m)

        0.6666666666666666
```

Figure 19: An example survey navigational action description for the action "jump to function body".

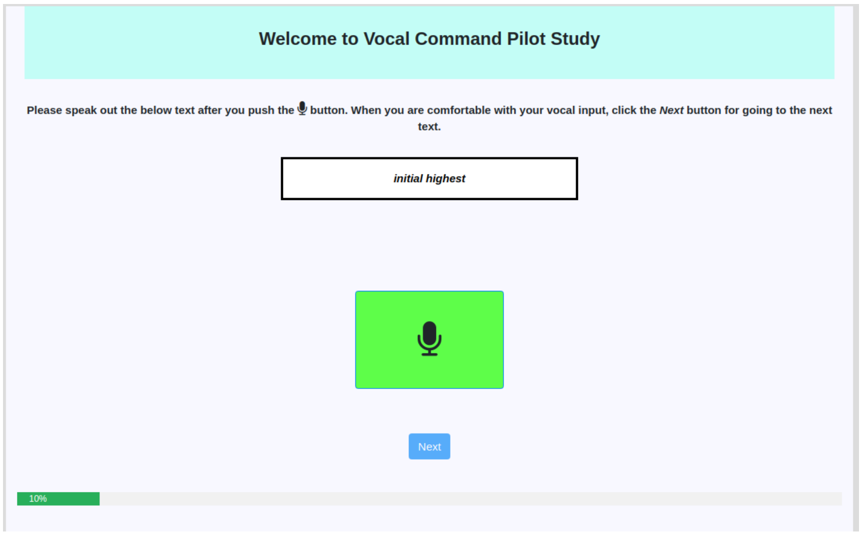
5.1.2. Voice-based study

This pilot study tried to understand our vocal commands and identifiers' transcription error rate, a common issue in voice-based systems. In the first step, we built our voice commands for code navigation based on the survey results (section 6.1) from the first pilot study. In this pilot study, we asked our participants to speak out the vocal commands. Our objective was to observe the common transcription mistakes for each vocal command. Initially, we wanted to only use the vocal commands for the study. However, we noticed that the command argument (for example, variable names) also creates transcription error. Besides, the combination of the command and its argument may produce unseen transcription mistakes (for instance, *initial highest* for the command *initial* and the variable *highest*). Therefore, we decided to put both command and variable together as the study inputs. Later in our tasks design (section 5.5), we chose the variables and functions name based on this pilot study.

We developed a web application[14] with using PHP programming language where

[14]`https://github.com/Pooya-Oladazimi/vocal-experiment`

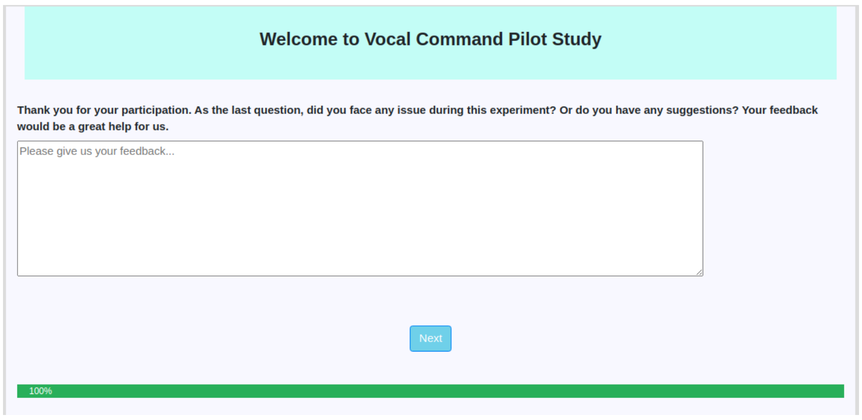
participants speak out the inputs (Figure 20). The study included twenty sample of command+argument (Table 3). At the end of the study, we asked our participants to provide the challenges they faced during the experiment (Figure 21). We aimed to improve our voice setup design with respect to participants' suggestions.


Welcome to Vocal Command Pilot Study

Please speak out the below text after you push the button. When you are comfortable with your vocal input, click the *Next* button for going to the next text.

initial highest

Figure 20: Voice pilot study. Participant clicks the green record button and then, specks out the input.


Welcome to Vocal Command Pilot Study

Thank you for your participation. As the last question, did you face any issue during this experiment? Or do you have any suggestions? Your feedback would be a great help for us.

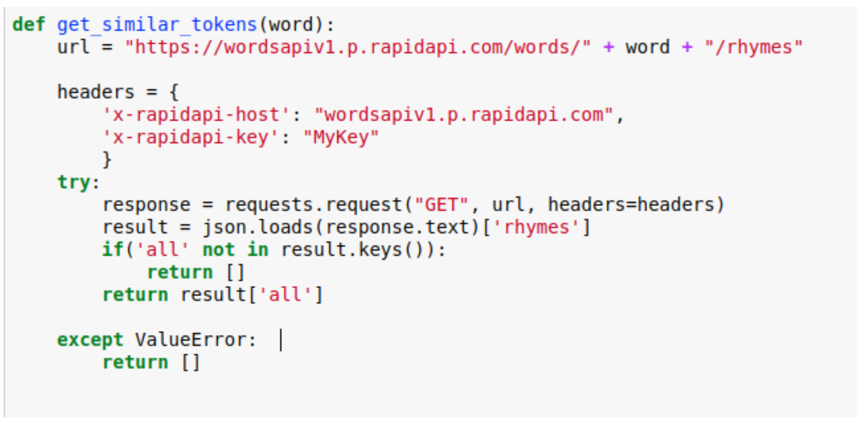
Please give us your feedback...

Figure 21: Voice pilot study. Participant can provide the challenges they faced during the experiment. Later, we used them to improve the voice setup design of this study.

List of vocal inputs	
1- Function tree	11- Documentation itemgetter
2- go to Function orange	12- Search add legend to plot in matplotlib
3- Initial highest	13- Next
4- Initial room	14- go back
5- Last topic	15- go to Function many
6- Last offset	16- Initial lowest
7- Change method fruit	17- go to Last ID
8- go to Function banana	18- go to Documentation boxplot
9- Change method plant	19- Function blueberry
10- Documentation randint	20- go to Initial total

Table 5: The list of the inputs we used in the voice pilot study.

There are some limitations to this vocal pilot study. We cannot put all the combinations of command+variable/function/API since it unreasonably increases the participants' experiment duration. Also, unlike variable/function names, API names cannot be customized by us. The latter is also true in a real case scenario where programmers can define their variables/functions, but it is cumbersome to change API names. In addition, due to the low number of participants, we might miss some of the common transcription mistakes for commands and arguments. Therefore, we utilized WordsAPI[15] to overcome these limitations. We used the Rhymes functionality of WordsAPI that provides words with a similar sound to the target word. Figure 22 shows the request structure in Python that we used to collect similar rhymes. We performed it for all the commands, variables, functions, and APIs which we used in the tasks (Section 5.5) of this study.



```python
def get_similar_tokens(word):
    url = "https://wordsapiv1.p.rapidapi.com/words/" + word + "/rhymes"

    headers = {
        'x-rapidapi-host': "wordsapiv1.p.rapidapi.com",
        'x-rapidapi-key': "MyKey"
        }
    try:
        response = requests.request("GET", url, headers=headers)
        result = json.loads(response.text)['rhymes']
        if('all' not in result.keys()):
            return []
        return result['all']

    except ValueError:
        return []
```

Figure 22: The code snippet which collect the similar words in term of rhymes for the provided word. We ran it for all the commands, variables, functions, and APIs exist in the tasks.

[15]https://www.wordsapi.com/

5.2. Phase0: Pre-experiment questionnaire; Understanding participants prior knowledge

Each participant fill in a questionnaire regarding his/her background knowledge at the first experimental session (Figure 18). We provided some questions (Table 6) to understand participants' prior knowledge and level of expertise in Python programming language, Programming, eye-tracker systems, voice recognition, Jupyter Notebook, and using keyboard shortcuts. The goal is to observe the possible effect of this prior knowledge on participants' performance in different setups. We used Google forms[16] to implement this phase. Each participant filled in this form in the first session of his/her experiment procedure.

Question	**Options**
How many years experience do you have in programming?	none, less than 1 year, between 1 year and 3 years, between 3 year and 5 years, More than 5 years
How many years experience do you have with Python programming language?	none, less than 1 year, between 1 year and 3 years, between 3 year and 5 years, More than 5 years
Do you have experience with the Jupyter Notebook?	No, Yes
Do you have industrial experience with Python programming? (Working experience in a company)	No, Yes
Do you have industrial experience in programming? (Working experience in a company)	No, Yes
which programming languages are you comfortable with? You can choose more than one.	Python, JavaScript, PHP, Java
Do you have experience with eye-tracking devices?	No, Yes
Do you have experience with voice recognition systems?	No, Yes
How do you rate your level of expertise in Python programming?	5-point scale
How do you rate your level of expertise in programming?	5-point scale
How do you rate your level of expertise in using the keyboard shortcuts? (generic to programming)	5-point scale

Table 6: The questions in the Phase0 of the experiment. The goal is to understand participants prior knowledge effect on results.

[16]https://www.google.com/forms/about/

5.3. Phase1: Wizard of Oz Approach

In phase1, we tell participants the exact navigational actions for understanding source codes, a common activity in code maintaining [12] (example: Table 7). This way, we can neutralize our participants' background knowledge, affecting the task completion time. Also, since the navigational commands are shared between participants, we can compare setups' performance in terms of time and accuracy to answer the second research question.

At the beginning of the experiment, we perform a sample task for participants as a part of the familiarization process. Then, participants performing two tasks for training (Figure 8). We do not start the actual experiment until the participants feel comfortable with their training sessions. Once participants are comfortable, the actual experiment comprising of three maintenance tasks is undertaken.

One challenge that we faced in this phase was in Voice-Mouse setup. The problem was that the speech recognition API could not recognize speeches when the experimenter and participants talked (reading the commands while participants were performing them). Therefore, we recorded all commands by the experimenter's voice on a mobile phone and played them via a headset device. This way, we make the experiment fair for all participants since they could hear the commands in the same tone of voice.

5.4. Phase2: Free Debugging Task

The Wizard of Oz approach in phase1 is suitable for observing the time and accuracy of the setups. However, it might not be suitable for measuring system usability and participants' perceived performance. The reason is, in phase1, participants are limited to do some navigational actions, and the outcome might have a bias toward the chosen commands by the experimenter. Therefore, in phase2, we perform a free-style experiment to mimic an actual code maintaining scenario. Minelli et al. [17] stated that developers, on average, spend 70% of their time on code understanding. Therefore, we ask participants to understand source codes by reading and performing their desire navigational actions. Participants have ten minutes for each task. The goal of phase2 is to answer the first research question and the usability comparison in the second research question. We perform setups' familiarization and participants' training exactly like phase1 based on the experiment flow in Figure 8.

5.5. Tasks

We designed six tasks for this study experiment. We designed them by utilizing open-source projects on Github and our knowledge in Python programming. We also designed three lists of navigational actions for phase1 of the experiment (Tables 7, 8, 9). We used the survey result (section 6.1) to give each navigational action frequencies in these lists. We gave more important navigational actions higher frequency. For example, we gave Online navigational actions (Search, API documentation, and Next search result) higher frequency since it obtained a higher importance score in the survey. Besides, we named the variables and functions based on the vocal pilot study (section 5.1.2) content.

- Task1 (phase1): We adopted a GitHub repository[17] for this task. The task includes Fibonacci, Factorial, and circle area algorithms. It contains 34 navigational actions (Table 7).
- Task2 (phase1): We adopted a GitHub repository[18] for this task. It contains four sorting algorithms (Topological sort, Bead sort, Bubble sort, and Gnome sort). It contains 41 navigational actions (Table 8).
- Task3 (phase1): We adopted two GitHub repositories[19][20] for this task. It contains some algorithms related to string and array processing (Longest common subsequence, Closest pair of points, Euclidean distance, Column based sort, and All pair of elements in a list). It contains 42 navigational actions (Table 9).
- Task4 (phase2): A python code that performs some statistical measures (Mean, Mode, Median, Variance, Absolute frequency, Cumulative frequency, and Percentile) on students' scores in a course.
- Task5 (phase2): An image crawler that gets all images from a web page on HTTP protocol and stores them in a directory.
- Task6 (phase2): A python script that classifies persons to their ideology based on their speech content.

[17] `https://github.com/whojayantkumar/Python_Programs`
[18] `https://github.com/TheAlgorithms/Python/tree/master/sorts`
[19] `https://github.com/TheAlgorithms/Python/tree/master/divide_and_conquer`
[20] `https://github.com/TheAlgorithms/Python/tree/master/dynamic_programming`

Task1 commands list
The code in front of you contains Fibonacci, Factorial, and circle area algorithms. The algorithms input are string characters in Unicode format. In this task, we want to read the code together and understand how the code uses mentioned algorithms. To achieve the goal, please follow the below navigational steps:
• Scroll down to the “Start” cell.
• in the start cell, first line: check the variable “topic” where it was last modified.
• Go back to the start cell
• in the start cell, first for loop: check the variable “total” initial value.
• Go back to the start cell
• in the start cell, second for loop: Check documentation of the keyword “ord”.
• in the start cell, second for loop: check the variable “fix” initial value.
• Go back to the start cell
• In the start cell, second for loop: check the function definition of “banana”
• In the function “banana” definition: check the function definition of “blueberry”
• Go back to the function definition of “banana”
• Go back to the start cell
• in the start cell, last line: check the variable “highest” where it was last modified
• Go back to the start cell
• in the start cell, last line: check the function definition of “orange”
• In the function “orange”, use the search functionality to ask: “add item to list in python”
• check the next search result
• Go back to the function “orange”
• Go back to the start cell
• in the cell 1, first line: check the variable “Id” where it was last modified
• Go back to the cell 1
• in the cell 1, in the for loop: check the function definition of “many”
• Use the search functionality to ask: “Factorial implementation in python”
• check the next search result
• check the next search result
• check the next search result
• Go back to the cell 1
• in the cell 1, in the for loop: check the documentation for the API “sqrt” which is part of math library
• in the cell 2, first line: check the function definition which modified the variable “red”
• Go back to the cell 2
• Use the search functionality to ask: “power two in python”
• check the next search result
• Go back to the cell 2
• in the cell 2, first while loop: check the documentation for the keyword “len”.

Table 7: List of commands in the Task1 of phase1.

Task2 commands list

The code in front of you contains four sorting algorithms. You can see the dataframe which shows the input and output for each algorithm at the cell 5 output. In this task, we want to read the code together and understand these four sorting algorithms. To achieve the goal, please follow the below navigational steps:

• Scroll down to the "Start" cell.

• in the start cell, first line: check the variable "monkey" initial value

• Go back to the start cell

• in the start cell, first line: check the function definition of "many"

• In the function "many", Check documentation of the keyword "range", as you can see in the for loop

• Go back to the start cell

• in the start cell, line two: check the variable "fix" where it was last modified

• Go back to the start cell

• in the start cell, line two: check the variable "offset" where it was last modified

• Go back to the start cell

• in the cell 1, line one: check the function definition of "mushroom"

• Go back to the cell 1

• in the cell 1, line four: check the function definition which modified the variable "down" • Go back to the cell 1

• in the cell 2, line one: check the function definition of "tree"

• In the function "tree", use the search functionality to ask: what does assert do in python

• check the next search result

• check the next search result

• Go back to the function "tree"

• in function "tree": check documentation of the keyword "zip" , as you can see in the for loop

• In the function "tree", use the search functionality to ask: "what is enumerate in python"

• check the next search result

• check the next search result

• Go back to the function "tree"

• Go back to the cell 2

• in the cell 2, line one: check the variable "up" where it was last modified

• Go back to the cell 2

• in the cell 3, line one: check the function definition of "orange"

• Go back to the cell 3

• In the cell 3, check documentation of the keyword "str"

• in the cell 4, use the search functionality to ask: "how to define dataframe in python"

• check the next search result

• Go back to the cell 4

• in the cell 4, line 3, check documentation of the keyword "sorted"

• in the cell 4, use the search functionality to ask: " get a dictionary keys in python "

• check the next search result

• check the next search result

• check the next search result

• Go back to the cell 4

• in the cell 4, first line: check the variable "lowest" initial value

• Go back to the cell 4.

Table 8: List of commands in the Task2 of phase1.

Task3 commands list

The code in front of you contains a number of algorithms related to string and array processing. In this task, we want to read the code together and understand the algorithms objectives. To achieve the goal, please follow the below navigational steps:

• Scroll down to the “Start” cell.
• in the start cell, first line: check the variable “fix” initial value
• Go back to the start cell
• in the start cell, first line: check the variable “identify” where it was last modified
• Go back to the start cell
• in the start cell, third line: check the variable “room” initial value
• Go back to the start cell
• in the start cell, last line: check the function definition of “orange”
• In the function “orange”, Check documentation of the keyword “max"
• In the function “orange”, Check documentation of the keyword “float”
• Use the search functionality to ask: “finding closest data points in python”
• check the next search result
• check the next search result
• Go back to the start cell
• in the cell 1, last line: check the function definition of “tree”
• In the function “tree”, Check documentation of the keyword “sorted"
• In the function “tree”, Use the search functionality to ask: “column based sort in python”
• check the next search result
• Go back to the cell 1
• in the cell 2, last line: check the variable “up” where it was last modified
• Go back to the cell 2
• in the cell 2, last line: check the function definition of “mushroom”
• In the function “mushroom”, Use the search functionality to ask: “longest common subsequence in python”
• check the next search result
• check the next search result
• Go back to the cell 2
• in the cell 3, first line: check the variable “total” initial value
• Go back to the cell 3
• In the cell 3, first line: check the function definition of “fashion”
• In the function “fashion”, check the function definition of “banana”
• In the function “banana”, Check documentation of the keyword “range"
• In the function “banana”, Use the search functionality to ask: “finding all possible combination in a python list”
• check the next search result
• check the next search result
• check the next search result
• Go back to the cell 3
• in the cell 3, last line: check the variable “identify” where it was last modified
• Go back to the last point
• in the cell 4, Use the search functionality to ask: “get a dictionary keys in python”
• check the next search result
• check the next search result
• Go back to the cell 4

Table 9: List of commands in the Task3 of phase1.

5.6. Evaluation

We utilize both summative and formative measures for the evaluation. We utilize formative measure (user subjective feedback) because some studies showed that, although an interaction technique might be slower, the user feels more comfortable. For instance, in a comparison between the Dynamic and Static coloring approaches in gaze-click, the Static coloring was faster than the Dynamic coloring setup, but users felt better during using the Dynamic coloring [15]. Our formative evaluation includes:

- *System Usability Score (SUS)* [6]: We use ten questions. Participants answer to a 5-scale scoring for each question from Strongly Disagree to Strongly Agree. Figure 23 shows the standard for evaluating the SUS score. We calculate the SUS score for each setup for each participant this way:
  - X = Σ (odd numbered question score - 1)
  - Y = Σ (5 - even numbered question score)
  - SUS = (X + Y) * 2.5
- *7-scale Perceived Performance*: We ask our participants to answer a 7-scale question for these categories:
  - *Comfort*: How much comfortable was this setup to use?
  - *Speed*: How fast could you perform the tasks in this setup?
  - *Learnability*: How much difficult was it for you to learn this setup?
  - *Accuracy*: How much were you accurate in performing your tasks in this setup?
- *National Aeronautics and Space Administration-Task Load Index (NASA-TLX)* [10]
- *short interviews for asking participants feedback and opinions regarding setups*:
  - What are the advantages and disadvantages of using this setup for code navigation?
  - What do you suggest to improve this setup?
  - How do you rank your favorite setup from one to three? Why?

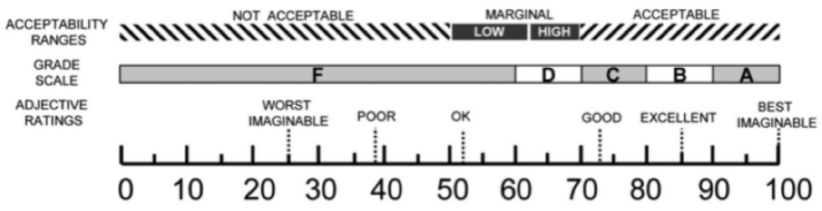


Figure 23: System Usability score measurement standard [6].

Our summative evaluation includes:

- *Time* : We tend to understand which setup demands less time for code navigation interactions. Besides, we want to understand what is the most proper setup for each navigation actions. For instance, CodeGazer showed that the gaze is suitable for jumping to a function body.
- *Accuracy* : We measure the number of errors that participants make in different setups while performing tasks. We aim to understand which setup raises the mistake chance. Each setup has different mistake types. We defined the types of mistakes we measure in this study in Table 10.

Setup	**Mistake Type**
Mouse+Keyboard	- Wrong shortcut - Typo in the target name
Gaze+Keyboard	- Choosing wrong color - choosing wrong navigational action
Voice+Keyboard	- Transcription mistake - Wrong command - Wrong target name

Table 10: The different types of mistake that can happen in each setup.

5.7. Experiment Procedure

Each participant participates in six experimental sessions on six different days (three sessions for each of phase1 and phase2). At the beginning of the first session, participants fill in the phase0 questions (section 5.2) and the user consent form. In each session, participants experiment with one setup. Also, we follow the workflow explained in section 4.2.3 of this thesis for each session. For Voice-Mouse and Mouse-Keyboard setups, we provide the printed list of navigational commands for participants that they can use during the entire experiment. We record the entire participant's performance in each session with a screen recorder application on windows 7. At the end of each session, participants answer the SUS, Nasa-TLX, Perceived Performance, and interview questions (section 5.6).

6. Results

In this chapter, we present our study results. First, we present the pilot study results, which we performed before the main experiment. After, we present our main experiment results, which we analyzed for each phase of this study experiment.

6.1. Survey Results

The survey had 39 participants (F=6, M=30, Other=3). Figure 25 and Figure 24 shows participants' level of expertise in Python programming and Programming. Based on the results, all the navigational actions in this study are more than moderately important and frequent (Figure 26, Figure 27). Also, Online Navigational actions (Search, Next result, API Documentation) obtained higher importance and frequency than Offline ones.

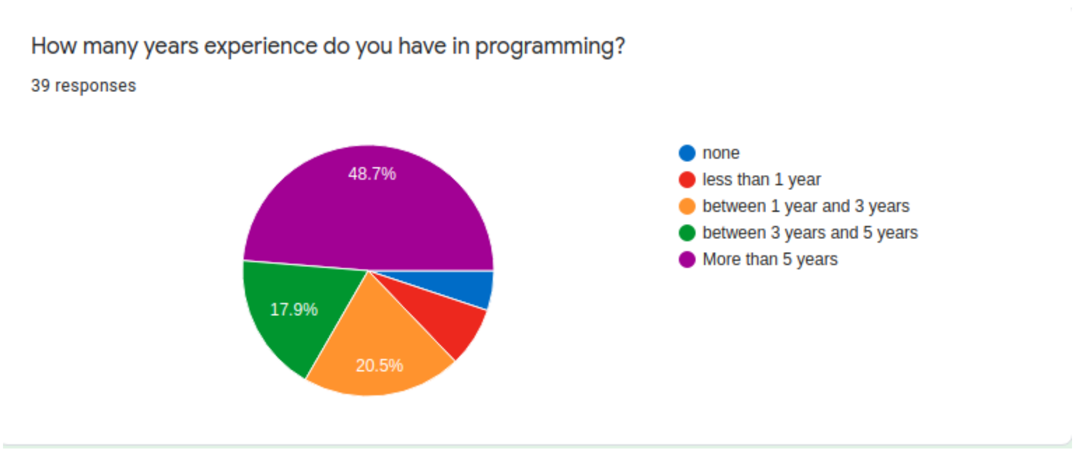


Figure 24: Result for years of experience in programming.

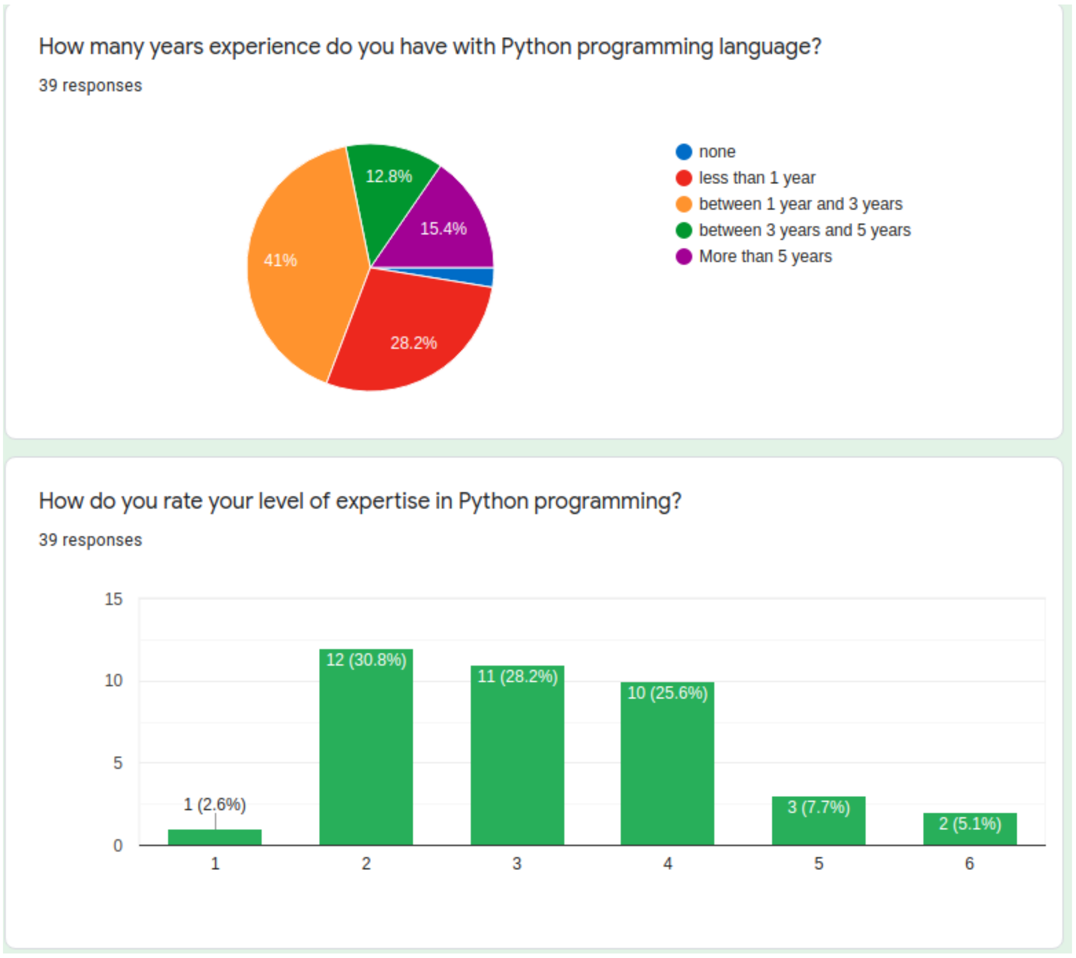


Figure 25: Result for years and level of expertise with Python programming.

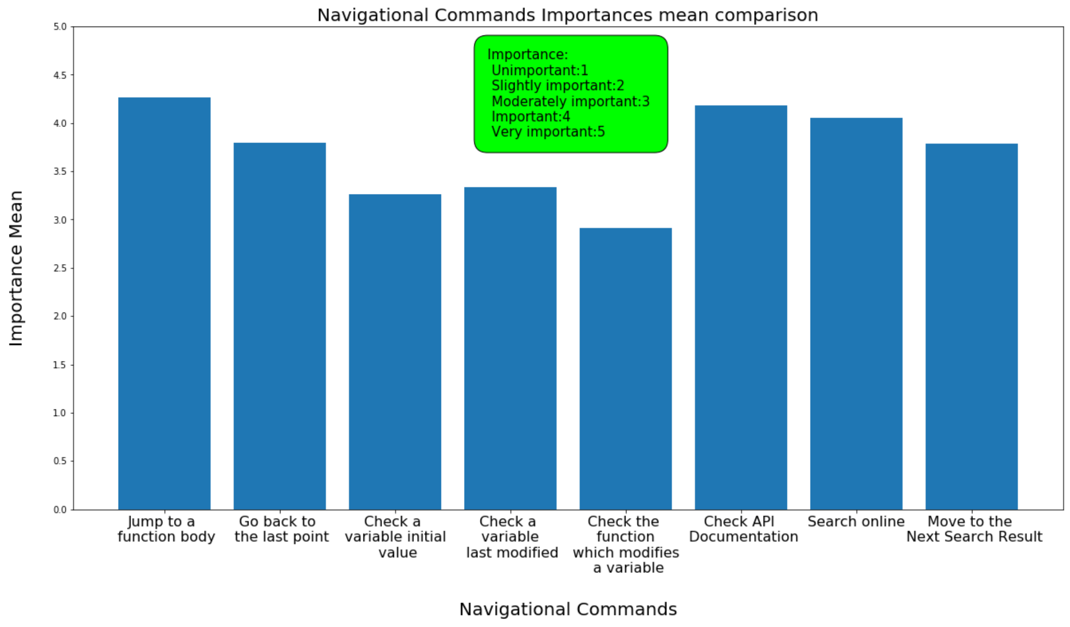


Figure 26: Average importance score for each navigational actions. Online navigational actions obtained higher importance score compare to offline ones.

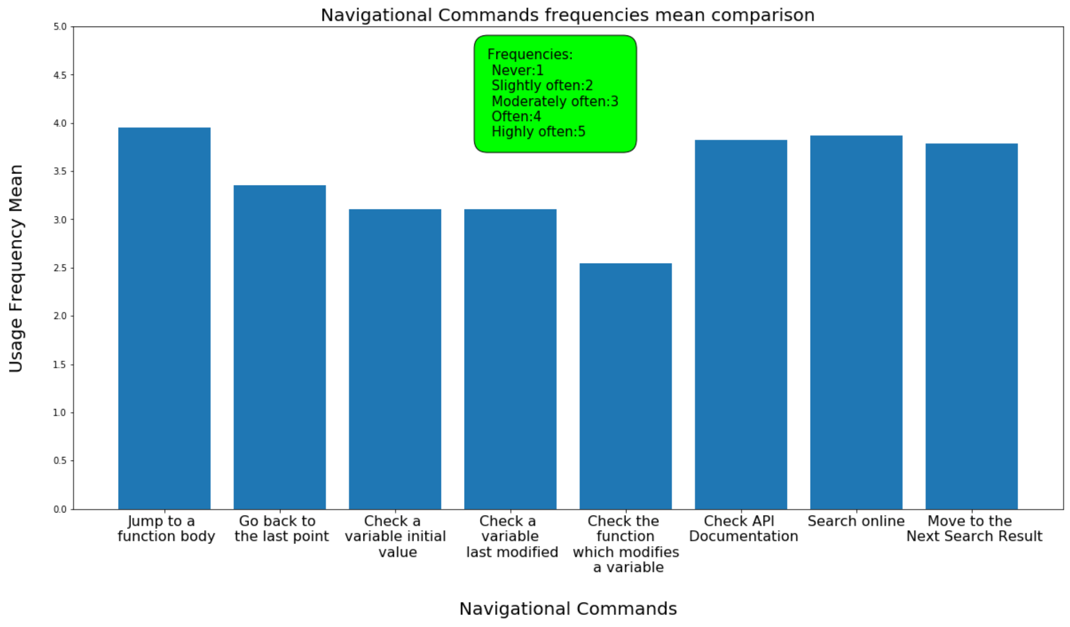


Figure 27: Average frequency score for each navigational actions. Online navigational actions obtained higher frequency score compare to offline ones.

To find the proper vocal command for each navigational action, we tokenized participants' suggestions based on white space to calculate the term frequencies. We considered three factors in choosing the proper term(s) for vocal commands: 1- Higher frequency, 2- Human understandable, and 3- Reflects the navigational command's intention as much as possible. We also tried to keep the command length to one or two words. Here are the results:

Jump to function body: We chose the token *function* as the command (Figure 28). The combination of *go* and *to* has a better frequency, but the *function* token reflects the intention behind this navigational command better, which makes it easier to remember and use.

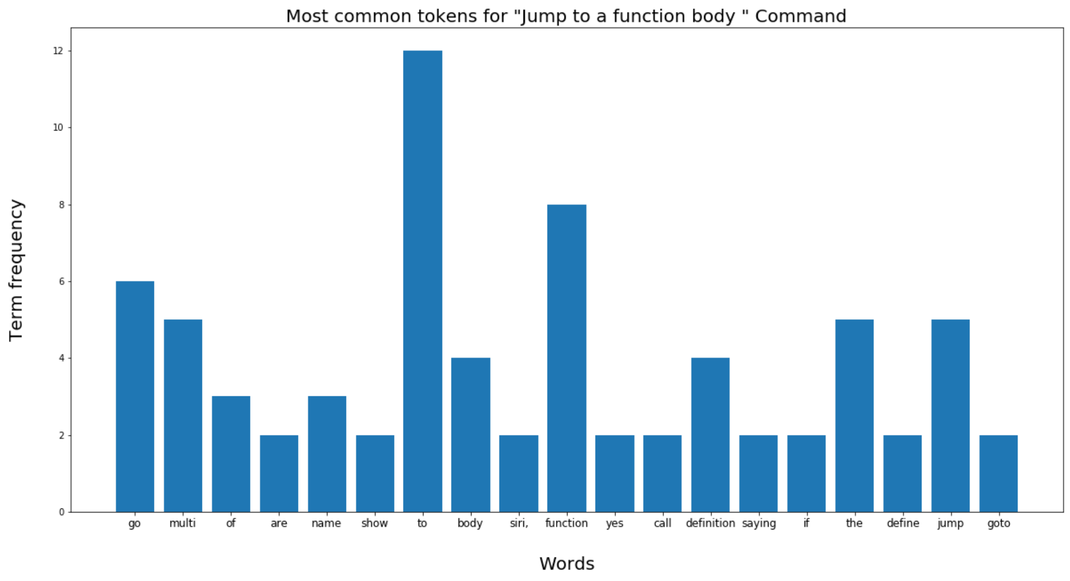


Figure 28: Term frequency for tokens based in participants suggestions for the *Jump to function body* navigational action.

Check a variable initial value: We chose the token *initial* as the command (Figure 29). The token *initial* does not have a higher frequency compared to others. However, it is short and reflects the intention better, which makes it easier to remember.

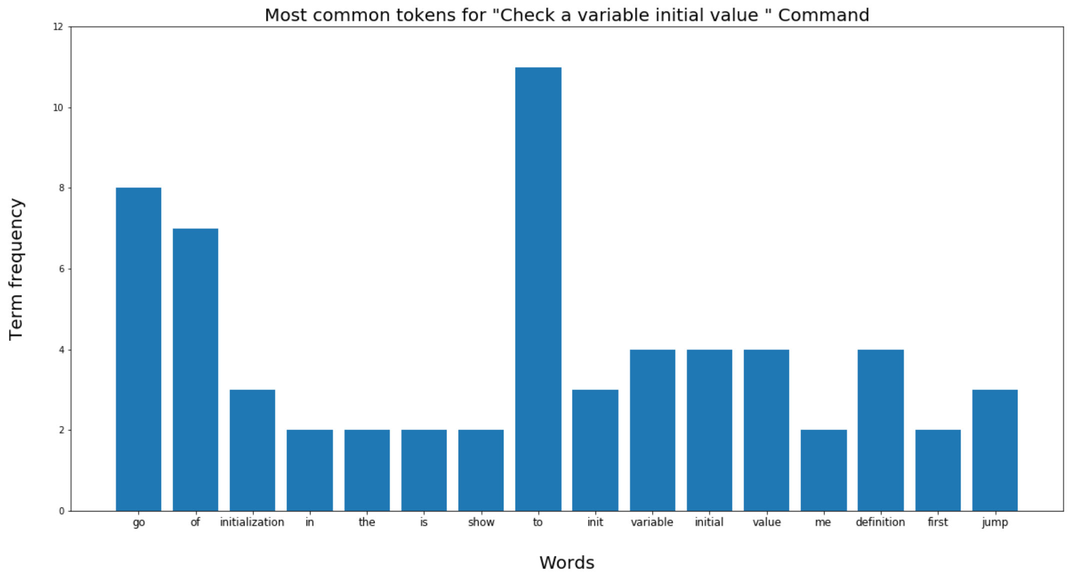


Figure 29: Term frequency for tokens based in participants suggestions for the *Check a variable initial value* navigational action.

Check a variable last modified: We chose the token *last* as the command (Figure 30). The token *last* has a higher frequency and also reflects the navigational action intention.

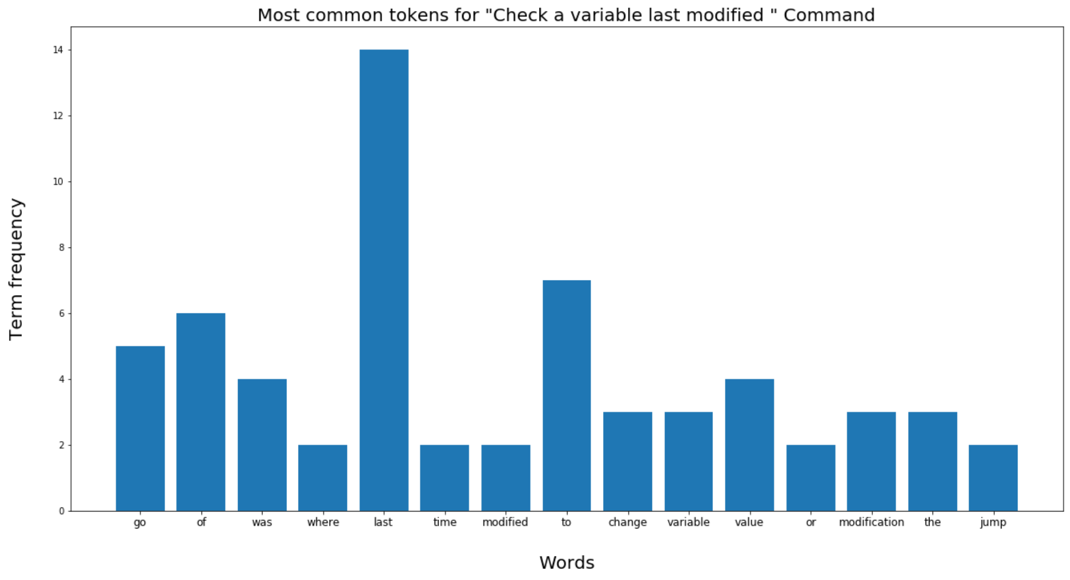


Figure 30: Term frequency for tokens based in participants suggestions for the *Check a variable last modified* navigational action.

Check the variable modifier function: We chose the tokens *change method* as the command (Figure 31). At first, we chose the token *change* as the command. However, we thought *change method* better reflects the intention behind the navigational command.

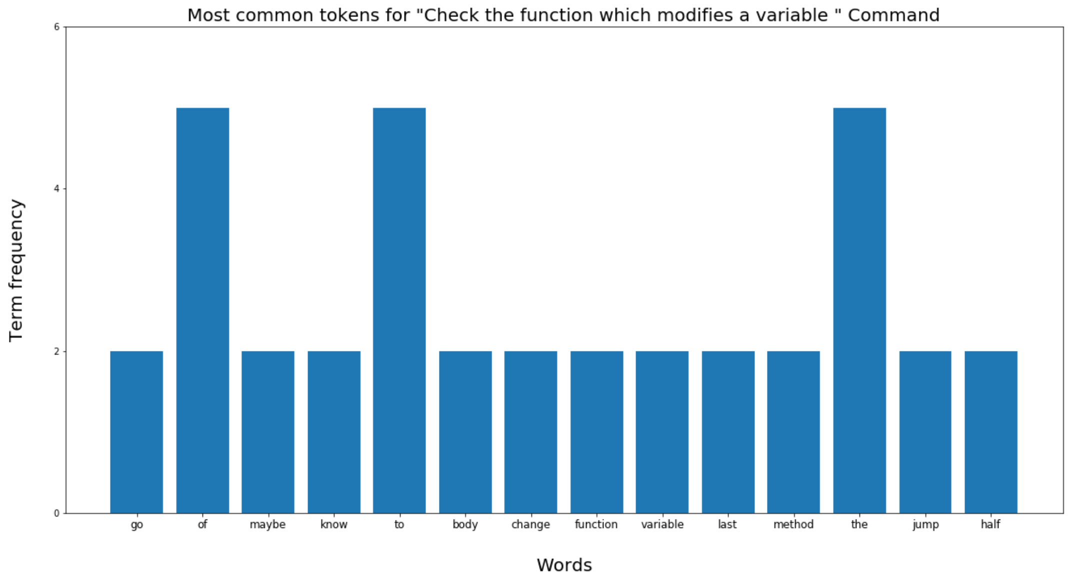


Figure 31: Term frequency for tokens based in participants suggestions for the *Check the variable modifier function* navigational action.

Go back to the last point: We chose the tokens *go back* as the command (Figure 32). The combination of the tokens *go* and *back* exactly reflects the navigational command intention.

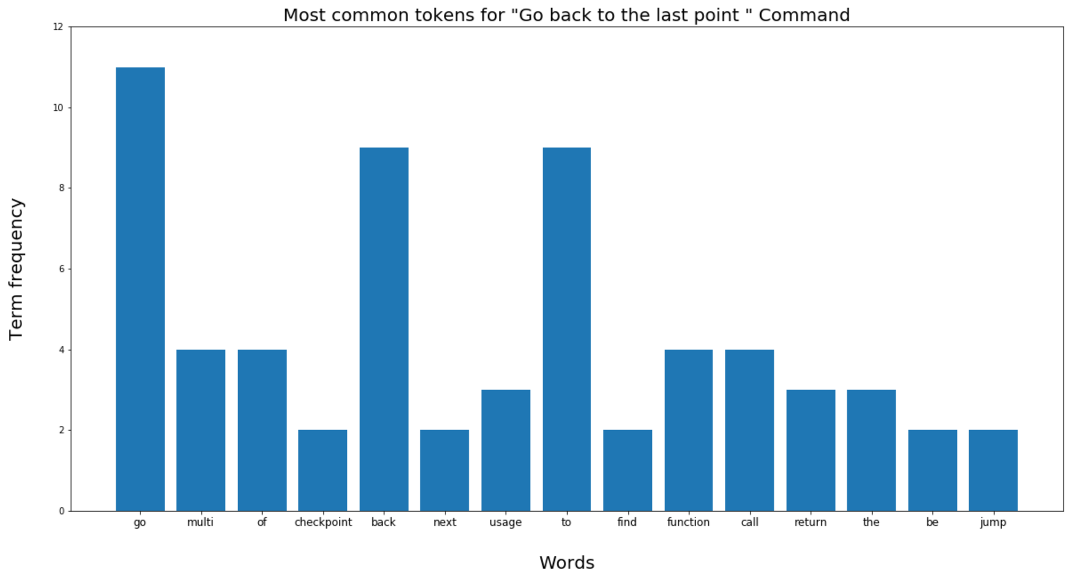


Figure 32: Term frequency for tokens based in participants suggestions for the *Go back to the last point* navigational action.

API documentation: We chose the tokens *documentation* as the command (Figure 33). Initially, we considered the token *doc* for the command since it is shorter than *documentation* and reflects the navigational action. However, the token *doc* has a similar rhythm to the word *duck*, leading to transcription errors in speech-to-text API.

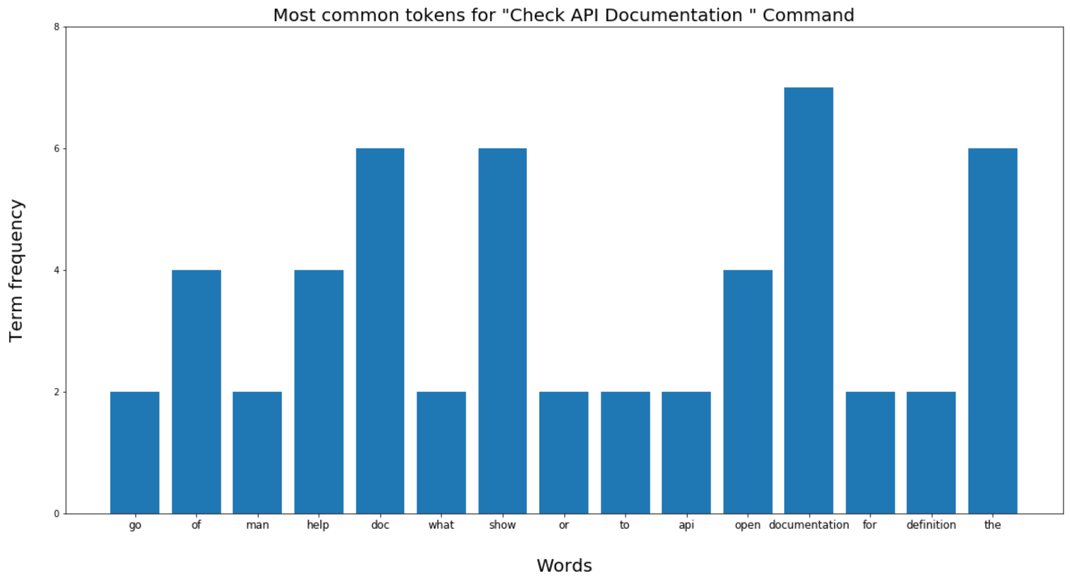


Figure 33: Term frequency for tokens based in participants suggestions for the *API documentation* navigational action.

Search online: We chose the tokens *search* as the command (Figure 34). It is a short command which completely reflects the navigational action purpose.

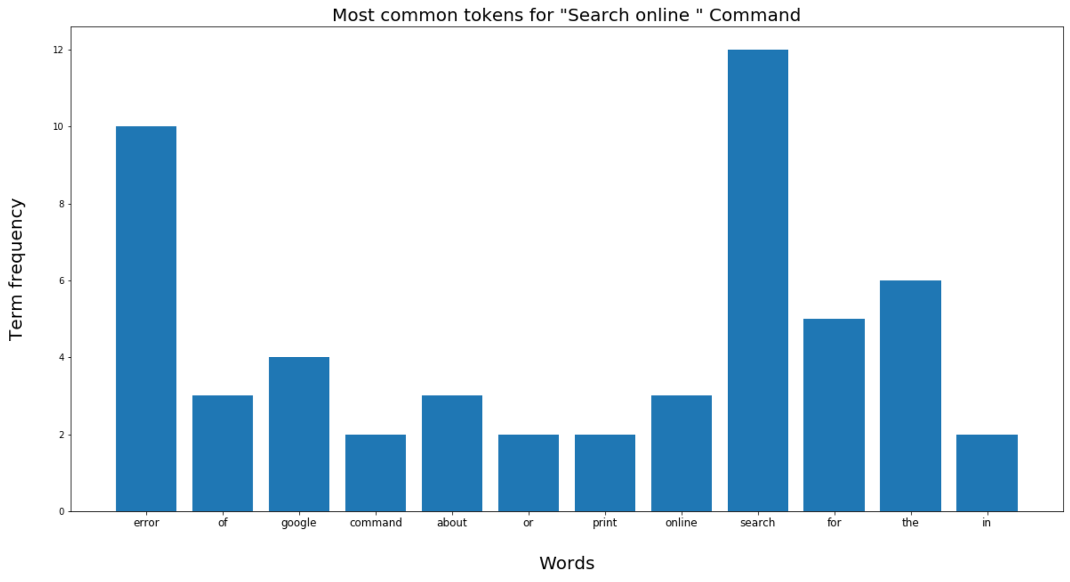


Figure 34: Term frequency for tokens based in participants suggestions for the *Search online* navigational action.

Move to the next search result: We chose the tokens *next* as the command (Figure 35). We could also combine the tokens *next* and *result*. Nevertheless, we decided to keep the command short.

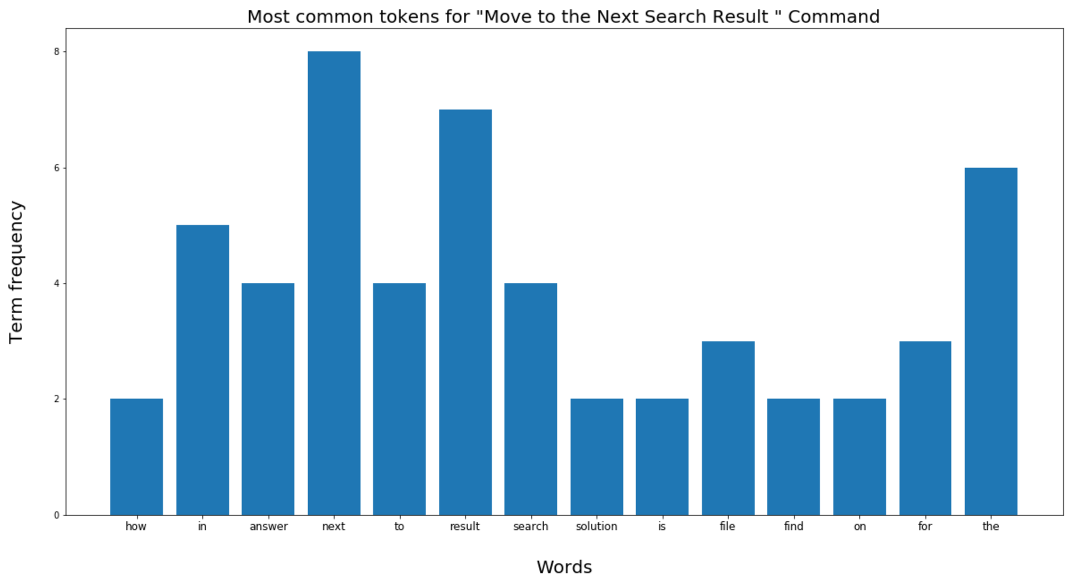


Figure 35: Term frequency for tokens based in participants suggestions for the *Move to the next search result* navigational action.

Commands importance and Python expertise level: At last, we tried to observe the possible relation between participants' level of expertise in python and navigational commands importance. We used the Pearson Correlation and utilized Levene's Test for evaluating whether the correlation is significant or not. Table 11 shows the evaluation results. A positive correlation shows participants with a higher level of Python expertise think the navigational action is important. In the opposite case(negative correlation), participants with lower Python expertise think the navigational action is important. One interesting observation is that Online navigational actions (documentation, Search, Next search result) have opposite correlation sign (negative) compare to others. This observation can indicate those novice participants care more about online search rather than code internal navigation. However, we cannot be sure about this observation since the result is not significant. The only significant result is "Jump to function body," which shows expert participants think checking a function definition is more important than others.

Action	Correlation	Significant
Jump to function body	+ 0.43	Yes
Check a variable initial value	+ 0.42	No
Check a variable last modified	+ 0.06	No
Check the variable modifier function	+ 0.06	No
Go back to the last point	+ 0.22	No
API documentation	- 0.14	No
Search online	- 0.07	No
Move to the next search result	- 0.02	No

Table 11: Correlation between survey participants Python level of expertise and navigational actions importance.

6.2. Voice Pilot Study Result

We tried to obtain some of the common transcription errors for our study's commands to reduce the transcription error rate. We invited participants via e-mail, social media posts, and direct invitation. The experiment was running for three weeks. We had 22 participants, which was less than we expected. All participants were non-native English speakers. We tokenized participants' speeches based on white space. We combined the results of this pilot study with WordsAPI to achieve higher accuracy. Table 12 shows the common transcription error for commands in this study.

Vocal Command	Transcription Error(s)
function	call shannon, hunting, punching, junction, punch, functions, fighting, phantom, find some, find
last	la, blast, fast, lost
initial	can you show, shell, alicia, any shared, show, any cell
change method	change methods, change metaphor, change metal, send method
Go Back	call beck, call back
Search	first
Next	-
documentation	recommendation, station, document, the documentation, beckenham station, communication

Table 12: Common transcription errors for command in this study obtained via voice pilot results.

6.3. Main Experiment Result

Due to the Coronavirus pandemic, we could not perform the experiment at the University of Koblenz-Landau laboratory. Therefore, The experiment took place in the experimenter flat. We did precaution actions for participants' safety. The experimenter wore a mask during the entire session. Besides, each participant had to sanitize his/her hands on the arrival. We also sanitized all the places after each participant and before participants arrived. Also, the experimenter was sitting two meters distance from the participant. We utilized a 24-inches monitor for the experiment. The eye tracker was MyGaze[21] which was installed at the bottom of the monitor (Figure 36). For the voice setup, we used the experiment laptop built-in microphone.

[21] http://www.mygaze.com/products/mygaze-eye-tracker/

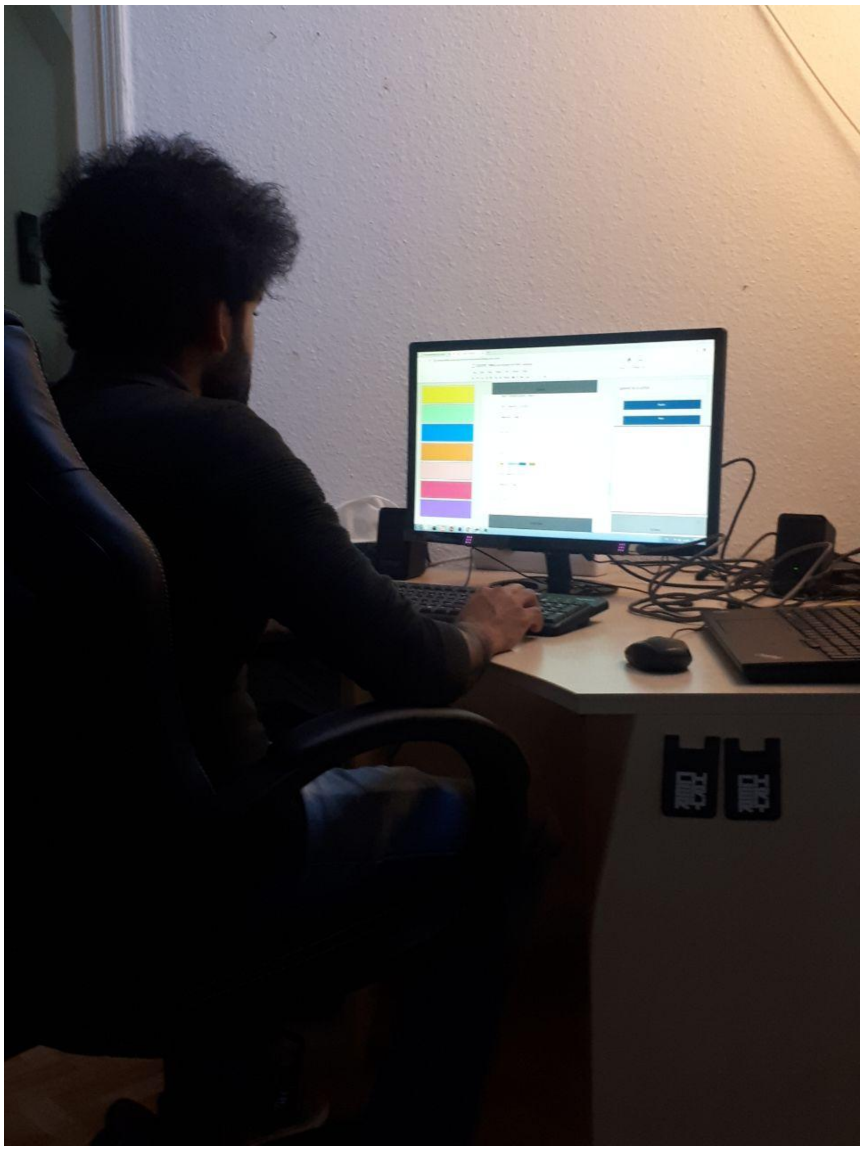


Figure 36: The participant was sitting in front of a 24-inches monitor which had the MyGaze eye tracker on the bottom. The laptop was at the participant right side which had a built-in microphone.

6.3.1. Participants

We utilized the phase0 results to gather participants' demographics. We had ten participants (F=4, M=6) who participated in this study's phase1 and phase2. Originally, we had 11 participants, but one of them could not continue his sessions due to personal reasons. All participants had experience with Python and Jupyter Notebook. Also, eight of them mentioned Python as their favorite programming language. Two of them had industrial experience with Python programming. Five of them had prior experience with an eye tracker. Six of them had prior experience with voice recognition systems.

6.3.2. Apparatus

We used MyGaze as the eye tracker, installed at the bottom of the monitor (Figure 37). We recorded the entire session with the Open Broadcaster Software[22]. Besides, we logged all participant's interaction in Comma-separated values (CSV) files for each session. The experiment laptop was a Lenovo Thinkpad with a Core i7 CPU (2.50 GHz) and 8G RAM. The operating system was Windows 7, and we used Google Chrome as the browser for Jupyter Notebook.

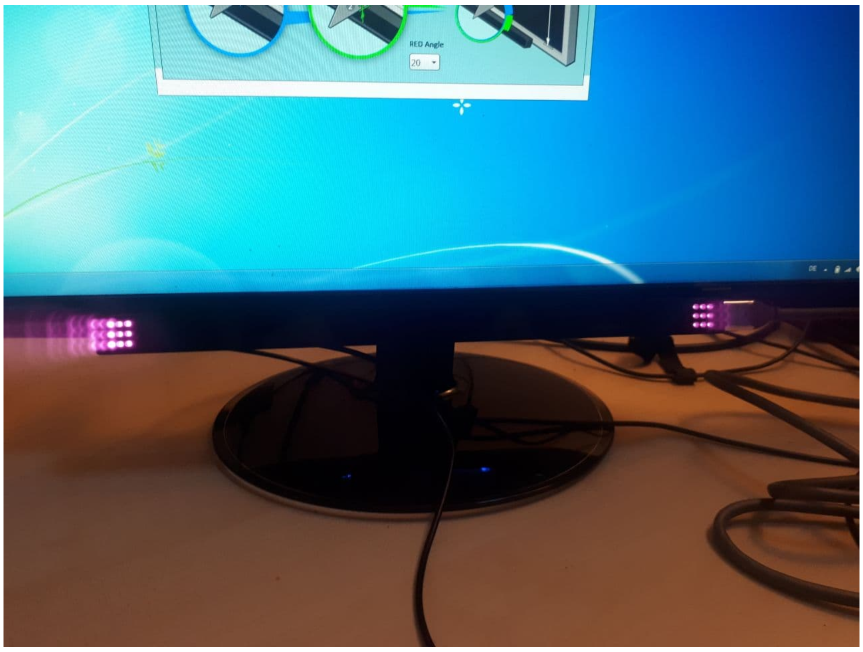


Figure 37: MyGaze eye tracker at the bottom of the experiment monitor.

6.3.3. Phase1 Results

In this phase, we did a controlled study, as described, is section 5.3 of this thesis. We used task1, task2, and task3 (section 5.5) for this phase. We were sitting behind the participant and playing the task commands (section 5.5) for participants via a mobile phone and a Bluetooth headset. We were also observing the participant's performance. We played the next command when the participant completed the current one. The participants could say the word "No" in case they could not perform a command. Also, they could say "Repeat" if they wanted to hear the command

[22]https://obsproject.com/

again. One challenge we had was with the transcription error in the Voice-Mouse setup. We told the participant to try each command three times at most in case of transcription error. After the third try, we move on to the next command.

The main goal of this phase is to evaluate the setups in terms of time and error rate. As mentioned in section 5.4, this phase might not be suitable for formative evaluation because of its controlled nature. However, we also gathered the formative feedback to compare them with phase2 results to observe the controlled study's effect on participants' opinions.

Task completion time: We define a task completion time as the duration from performing the first command to the last. Figure 38 shows the task completion time for each setup in this study. The Shapiro-Wilk test [19] shows that the results are drawn from a normal distribution. Also, the ANOVA test shows that the result is statistically significant ($F_{2,27}$ = 19.57, p-value = 0.000006), which indicates that the setup affects participants' time performance significantly. We also evaluated the setup effect on task completion time for each task separately (Figure 39). We found that setup also affects task1 ($F_{2,27}$ = 16.65, p-value = 0.000019), task2 ($F_{2,27}$ = 8.15, p-value = 0.001699), and task3 ($F_{2,27}$ = 7.50, p-value = 0.002553) completion time significantly.

Participants finished the tasks faster in the Mouse-Keyboard setup (μ = 491.92, M = 487.76). The slowest was the Gaze-Keyboard (μ = 635.32, M = 641.13). Voice-Mouse (μ = 513.96, M = 516.13) was close to the Mouse-Keyboard results. We performed a Post-hoc comparison with using Tukey HSD [4] for pair-wise comparison between setups. Both results of the Mouse-Keyboard and Voice-Mouse setups are significantly different from Gaze-Keyboard (p-value = 0.001). However, there is no significant difference between Mouse-Keyboard and Voice-Mouse (p-value = 0.64). This result can indicate that the Voice-Mouse setup is comparable with the traditional approach (Mouse-keyboard) in terms of task completion time. Therefore, we can say that adding Voice as an additional modality to the traditional approach does not affect programmers' time performance significantly.

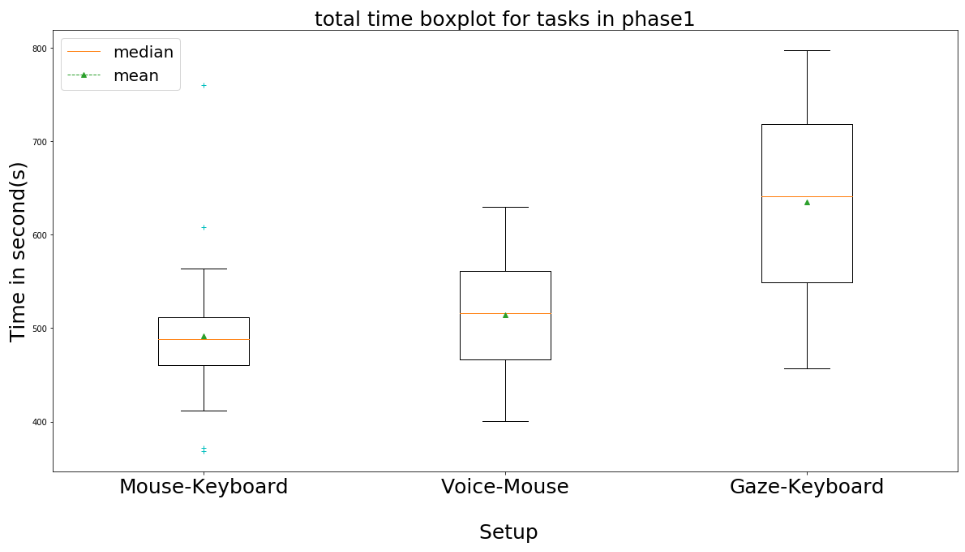


Figure 38: Total task completion time in phase1 for each setup.

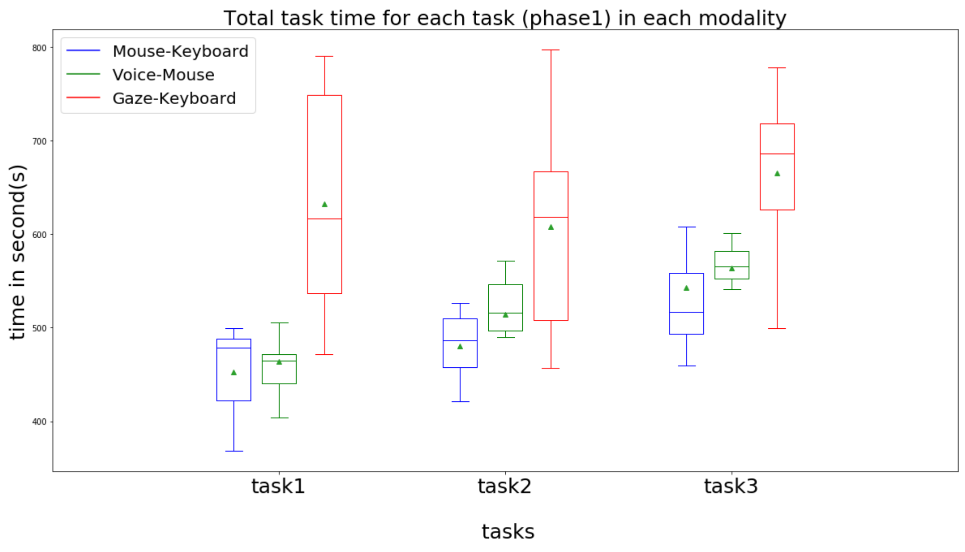


Figure 39: Total task completion time per tasks in phase1 for each setup.

Command Completion time: Although we tried to minimize idle times during performing a task, it still might affect the task completion time in the previous part. Therefore, in this part, we only consider the command completion time for each setup and task. This way, we can exclude idle time from participants' performance. Figure 40 shows the command completion time for each setup in this study. The Shapiro-Wilk test shows that the results are drawn from a normal distribution.

ANOVA test shows that the result is statistically significant ($F_{2,27}$ = 76.55, p-value = 7.473204e-12), which indicates that the setup affects participants' command completion time significantly. We also evaluated the setup effect on each task's command completion time separately (Figure 41). The setup also affects task1 ($F_{2,27}$ = 54.57, p-value = 3.267005e-10), task2 ($F_{2,27}$ = 23.29, p-value = 0.000001), and task3 ($F_{2,27}$ = 32.97, p-value = 5.650069e-08) command completion time significantly.

Participants performed the commands faster in the Mouse-Keyboard setup (μ = 136.52 , M = 129.97). The slowest was the Gaze-Keyboard (μ = 265.51, M = 140.17). Voice-Mouse (μ = 144.10, M = 140.17) was close to the Mouse-Keyboard results. We performed a Post-hoc comparison with using Tukey HSD for pair-wise comparison between setups. The Mouse-Keyboard and Voice-Mouse setups are significantly different from Gaze-Keyboard (p-value = 0.001). However, there is no significant difference between Mouse-Keyboard and Voice-Mouse (p-value = 0.78). This result can indicate that the Voice-Mouse setup is comparable with the traditional approach (Mouse-keyboard) in terms of command completion time. This result is consistent with the task completion time results in the previous part. It shows that the idle time in each task did not affect the overall results.

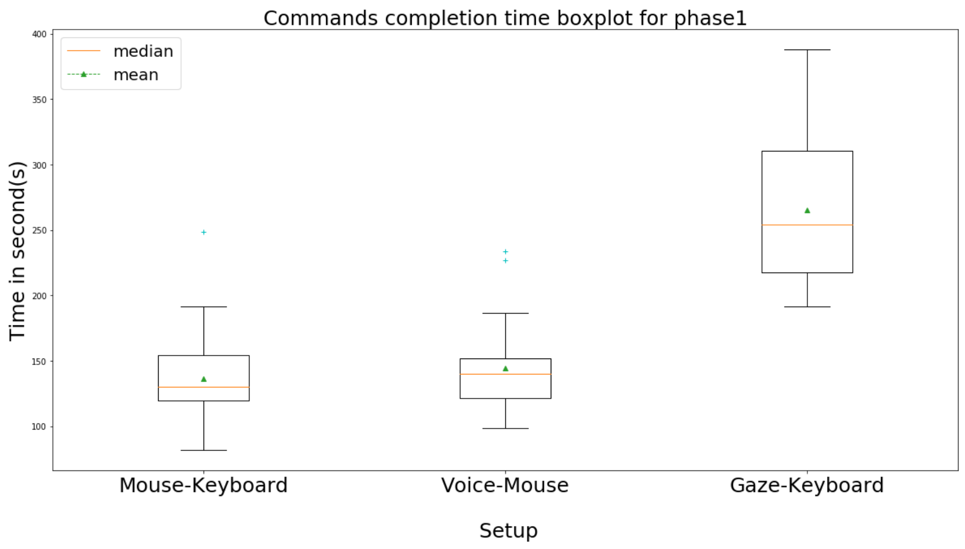


Figure 40: Command completion time in phase1 for each setup.

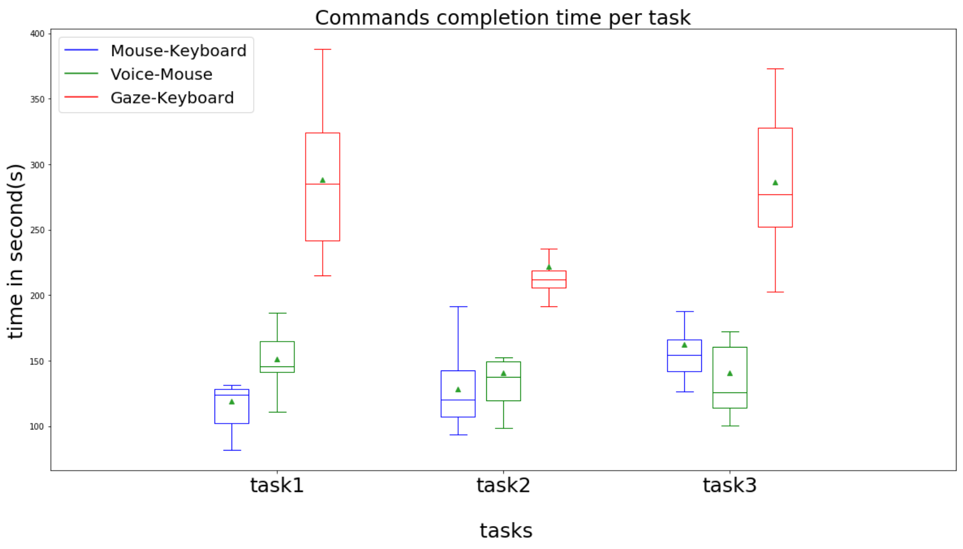


Figure 41: Command completion time per tasks in phase1 for each setup.

Command Completion time (Scroll excluded): We showed that setup significantly affects participants' time performance. We also showed that the Gaze-Keyboard was slower than other setups. However, this comparison might not be fair for the Gaze-keyboard since we already mentioned in section 4.4.1 that we slowed down scrolling for the Gaze-Keyboard setup. On the other hand, scrolling was not part of this study's main navigational actions (section 4.1). Therefore, we also tried to compare the setups in terms of command completion time by excluding the scrolling time. Figure 42 shows the command completion time for each setup in this study. The Shapiro-Wilk test shows that the results are drawn from a normal distribution. ANOVA test shows that the result is statistically significant ($F_{2,27}$ = 11.53, p-value = 0.00024), which indicates that the setup affects participants' command completion time significantly. We also evaluated the setup effect on command completion time for each task separately (Figure 43). The setup also affects task1 ($F_{2,27}$ = 6.86, p-value = 0.003), task2 ($F_{2,27}$ = 5.64, p-value = 0.008), and task3 ($F_{2,27}$ = 11.05, p-value = 0.00031) command completion time significantly.

Participants performed the commands faster in the Mouse-Keyboard setup (μ = 132.21 , M = 125.62). The slowest was the Gaze-Keyboard (μ = 183.53, M = 139.02). Voice-Mouse (μ = 140.84, M = 139.02) was close to the Mouse-Keyboard results. We performed a Post-hoc comparison with using Tukey HSD for pair-wise comparison between setups. The Mouse-Keyboard (p-value = 0.001) and Voice-Mouse (p-value = 0.002) setups are significantly different from Gaze-Keyboard. There is no significant difference between Mouse-Keyboard and Voice-Mouse (p-value = 0.72). Based on the result, we can say that our approach to slow down the scrolling did not affect the performance comparison between setups in terms of time.

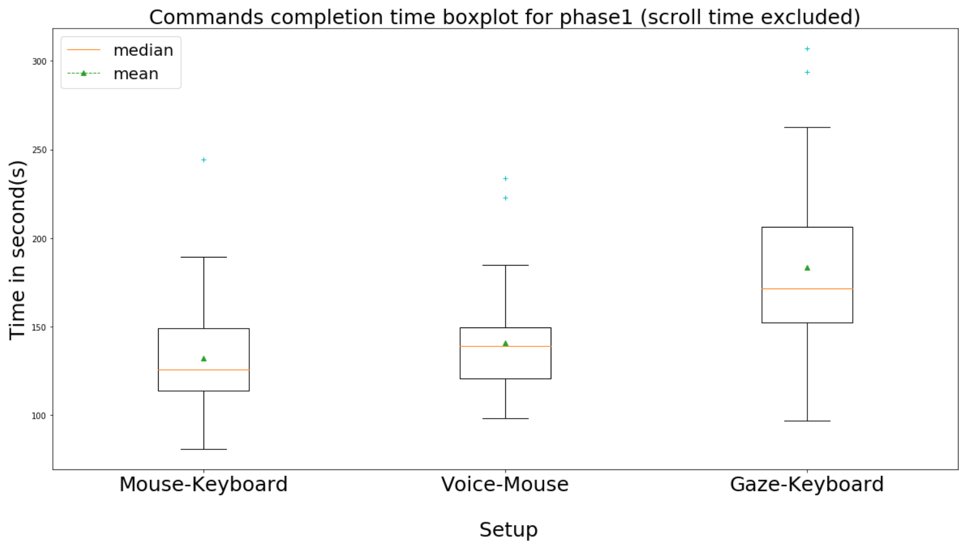


Figure 42: Command completion time (Scroll excluded) in phase1 for each setup.

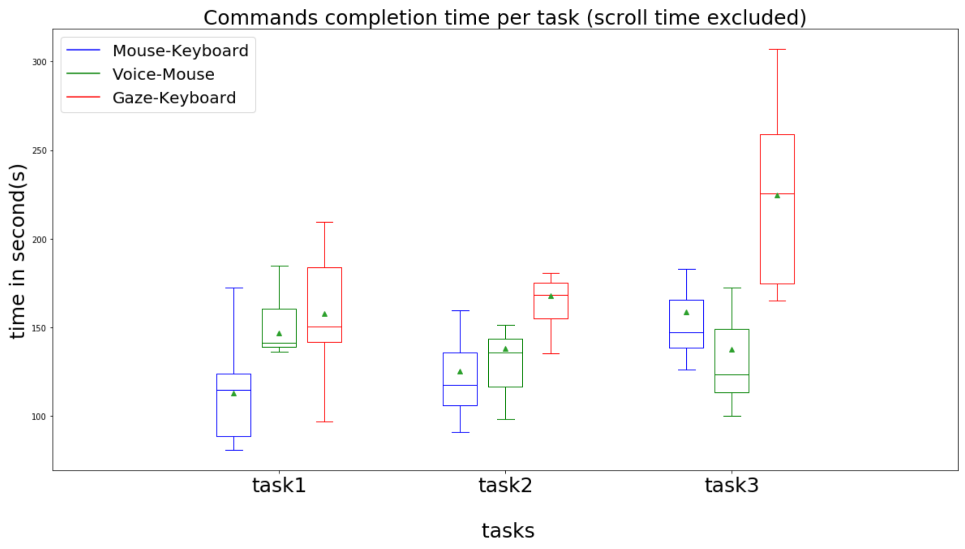


Figure 43: Command completion time (Scroll excluded) per tasks in phase1 for each setup.

Accuracy: We try to compare setups in terms of the number of mistakes that participants committed during phase1 of this study experiment. Figure 44 shows the accuracy of three setups in the phase1. The Shapiro-Wilk test shows that the results are drawn from a normal distribution. ANOVA test shows that the result is statistically significant ($F_{2,27}$ = 24.76, p-value = 7.786149e-07), which indicates that the setup affects participants' accuracy in code navigation. Mouse-Keyboard (μ = 1.63, M = 1) setup has the highest accuracy. Voice-Mouse was the least accurate (μ = 8.23, M = 7.5). Gaze-Keyboard was close (μ = 2.36, M = 2) to Mouse-Keyboard in terms of accuracy. Post-hoc comparison with using Tukey HSD shows that Voice-Mouse accuracy is significantly different from Mouse-Keyboard and Gaze-Keyboard (p-value = 0.001). However, there is no significant difference between Mouse-Keyboard and Gaze-Keyboard accuracy (p-value = 0.74). The result shows that adding a confirmation button (Ctrl key in this case) can reduce the Gaze-based approach Midas Touch issue and makes it comparable with the traditional approach in terms of accuracy. Voice-Mouse setup suffers from transcription error, which decreases accuracy in code navigation.

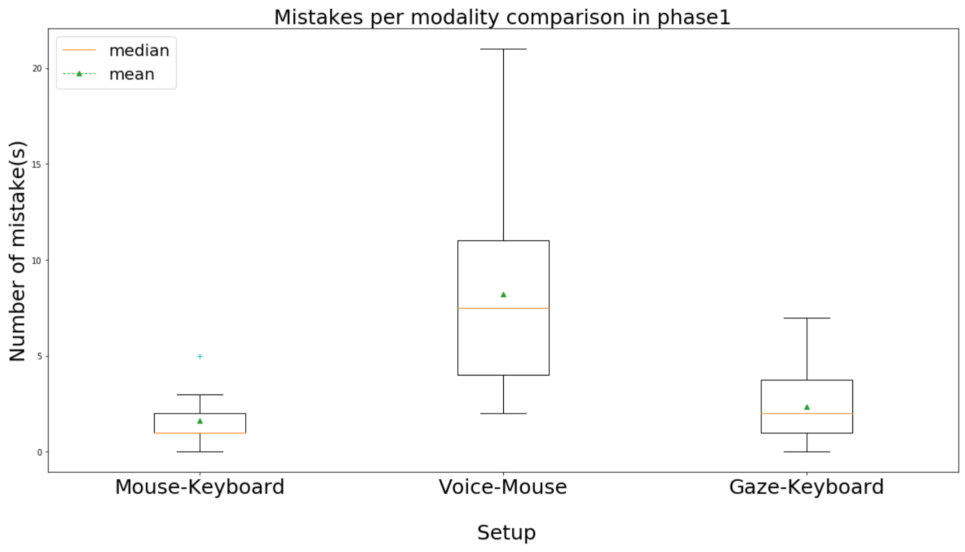


Figure 44: Mistakes committed by participants in each setup for phase1 of the experiment.

Transcription Error Prediction: We mentioned in section 5.4.1 that we added a prediction system based on our voice pilot study results to reduce transcription error in the Voice-Mouse setup. The results show that the prediction system had an almost 22 percent success rate (Figure 45). Step1 prediction is based on the voice pilot study result, and Step2 is based on WordsAPI. Although we could not resolve the issue with vocal transcription error, we reduced successfully. The result shows that simple steps like our prediction system can reduce transcription mistakes in code navigation with voice.

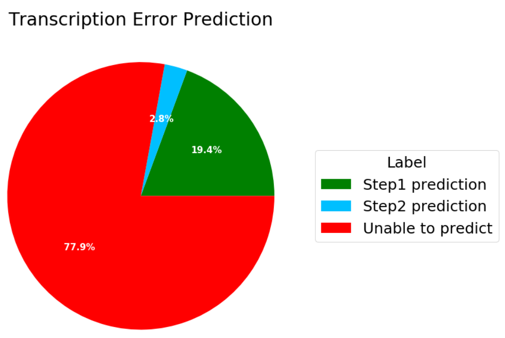


Figure 45: Transcription error prediction result in phase1.

Command-wise analysis: We performed a command-wise analysis in terms of command completion time (Figure 46) and accuracy (Figure 47). The result shows that the Voice-Mouse setup is better than others for the Search command. However, for the other Online navigational command (Documentation), Voice-Mouse was slowest. The reason might be that API names are not human language friendly. Consequently, the time penalty (because of command repetition) increases due to transcription errors. We can observe this effect in Figure 47, where Documentation has the highest mistake count for the Voice-Mouse setup. Gaze-Keyboard performed better in commands with no input argument (Go-back, Next result). One possible reason is that commands with no argument demand no target (identifier in the code) selection with the gaze.

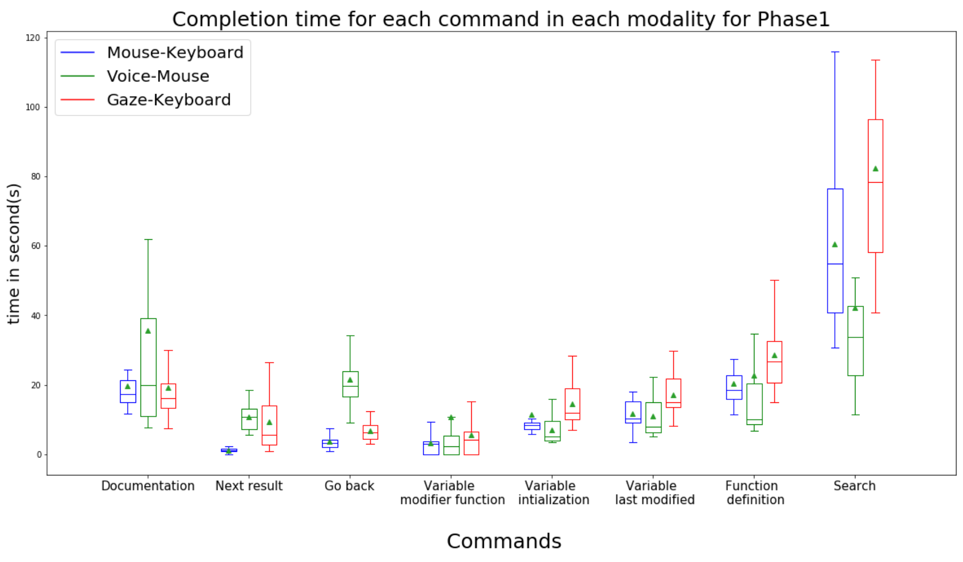


Figure 46: Command completion time per command in phase1 for each setup.

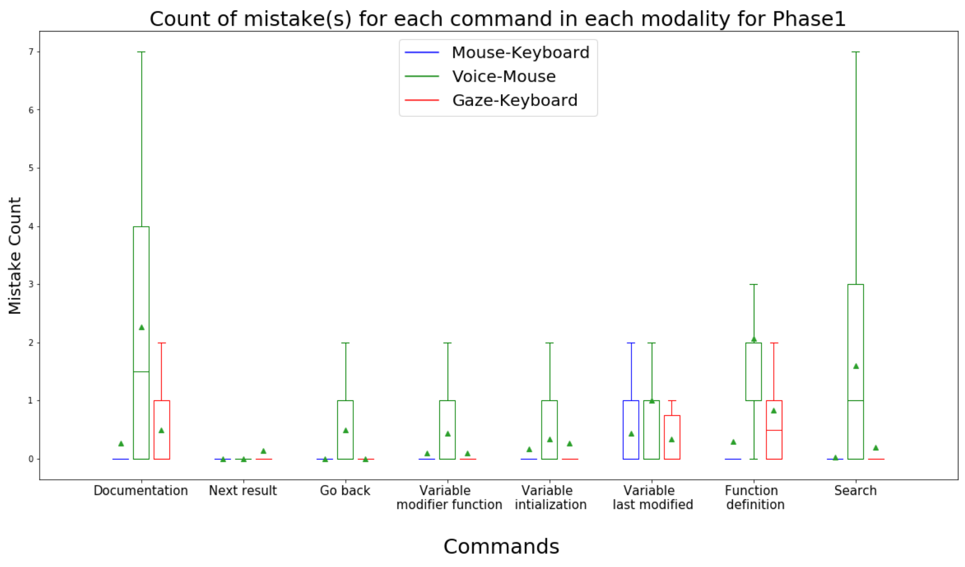


Figure 47: Mistake counts per command in phase1 for each setup.

System Usability Score (SUS): Figure 48 shows the setups SUS scores for phase1. Mouse-Keyboard setup obtained the highest SUS score (μ = 93, σ = 7.61), which maps to the "Excellent" based on the SUS standard (Figure 23). Voice-Mouse (μ = 83.75, σ = 17.76) obtained the "Good" score label. Gaze-Keyboard (μ = 66.75, σ = 25.30) obtained the "Ok" score based on the standard. Mouse-Keyboard and Voice-Mouse were "Acceptable," and Gaze-Keyboard was "Marginal (high) Acceptable" based on the acceptability standard shown in Figure 23. From the Grade Scale viewpoint, Mouse-Keyboard got "A," Voice-Mouse got "B," and Gaze-Keyboard obtained "D" scores.

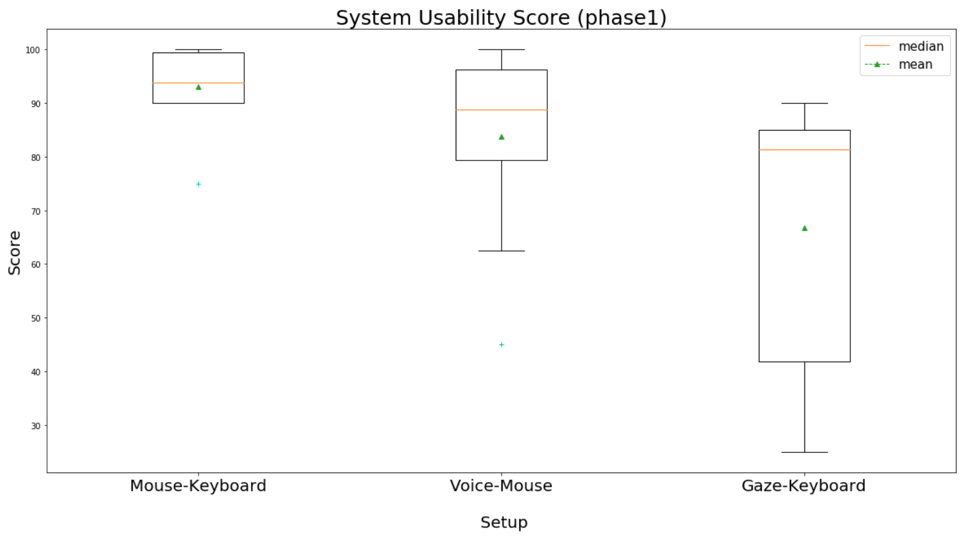


Figure 48: System Usability score in phase1.

NASA-TLX: NASA-TLX score shows the setup of physical-mental pressure through an overall score. Figure 49 shows the NASA-TLX for setups in this study. Mouse-Keyboard (μ = 33.13, σ = 18.88) setup has the lowest pressure among setups. Gaze-Keyboard (μ = 59.44, σ = 24.97) has the highest pressure, and Voice-Mouse (μ = 34.26, σ = 15.48) is in between the other setups in terms of task pressure.

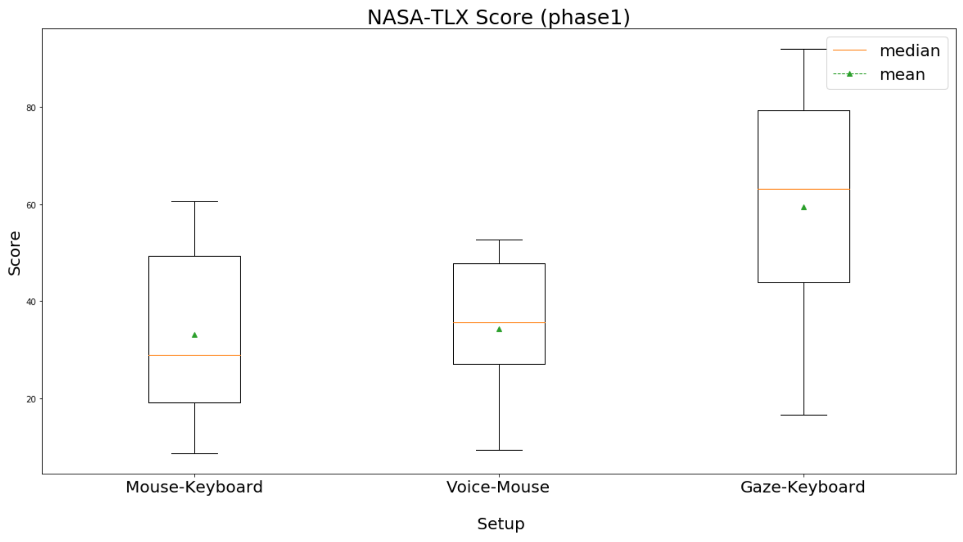


Figure 49: NASA-TLX score in phase1.

Perceived Performance: Perceived Performance shows participants' opinions of their code navigation performance under three setups in this study. Figure 50 shows the results for phase1.

- Learnability: Mouse-Keyboard (μ = 5.7) and Voice-Mouse (μ = 5.6) obtained very a close score in this category. Gaze-Keyboard got a lower score (μ = 4.7) in this category. The Gaze-Keyboard lower score is because the color-based code navigation was entirely new to all participants, making it more challenging to learn.
- Speed: Mouse-Keyboard (μ = 6.2) was the fastest setup in participants' opinions followed by Voice-Mouse (μ = 5.7) setup. Gaze-Keyboard (μ = 3.9) obtained the lowest score. The Gaze-Keyboard lower score is related to slow scrolling, and participants struggle with line selection. The latter was because of eye pupils fluctuation and gaze deviation, a common issue with eye tracker devices. The results are inline with our summative finding in terms of time.
- Accuracy: Participants felt the highest accuracy under Mouse-Keyboard (μ = 5.7) setup. One interesting finding is that Although Voice-Mouse was less accurate than Gaze-Keyboard in our summative Accuracy analysis, it obtained a higher score (μ = 5.4) than Gaze-Keyboard (μ = 4.7) from participants. The main reason was that participants challenge with line selection due to the pupils fluctuation and gaze deviation. This situation was especially more challenging for participants who wore glasses (three participants).
- Comfort: The most comfortable setup was Mouse-Keyboard (μ = 6.8) for participants. Voice-Mouse (μ = 5.8) was next in terms of comfort. This finding is because of participants' issues with transcription errors that made them repeat some commands a couple of times. The least comfortable was the Gaze-Keyboard (μ = 4.9) setup, mainly because of gaze deviation and multiple calibrations. For example, in one case, it took almost 20 minutes to find a proper sitting position for calibration because of participants' thick glasses. Also, some participants felt having dry eyes after performing tasks under the Gaze-Keyboard setup.

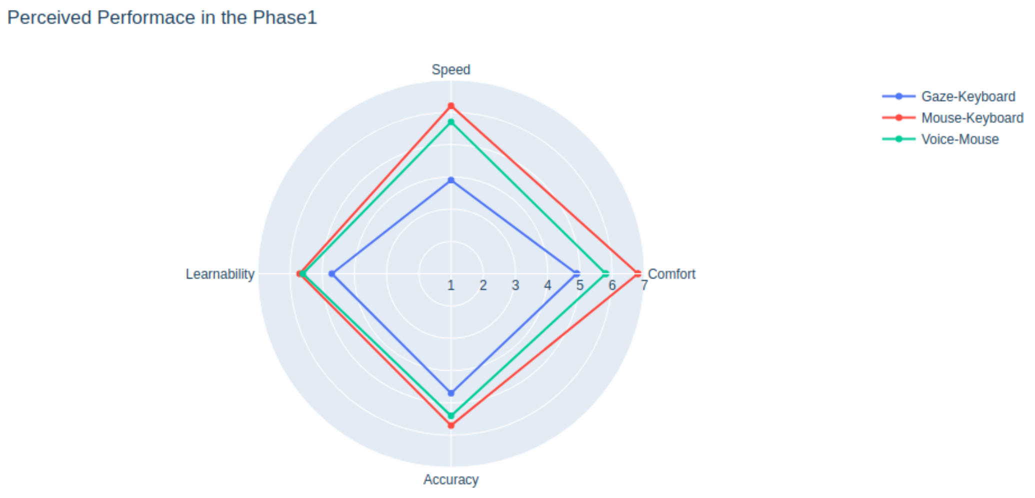


Figure 50: Participants Perceived Performance in phase1.

Participants' prior experience effect: We tried to evaluate the possible effect of participants' prior experience (obtained from phase0, section 5.2) on the phase1 results. First, we evaluated the effect of prior experience with eye-tracking on Gaze-Keyboard task completion time. Figure 51 shows the result. A T-test shows no significant difference between the two groups ($t(15)$ = 0.89, p-value = 0.38) that shows prior experience with eye tracker did not significantly affect our system result.

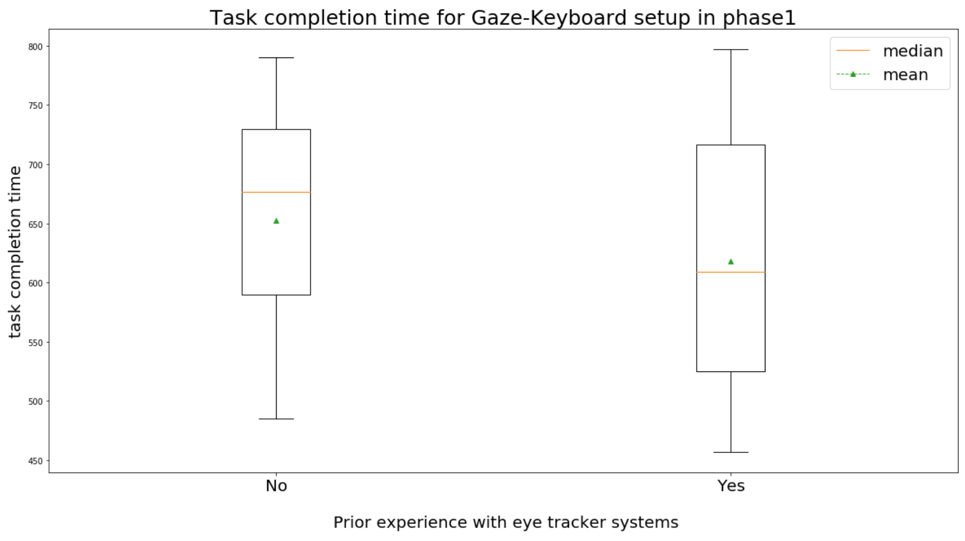


Figure 51: Time comparison for phase1 between participants with and without prior experience with eye tracking.

We also evaluated the effect of prior experience with speech recognition systems on Voice-Mouse task completion time. Figure 52 shows the results. A T-test shows there is no significant difference between the two groups ($t(18)$ = -0.36, p-value = 0.72). The results indicate that prior experience with speech recognition systems did not affect our system result significantly.

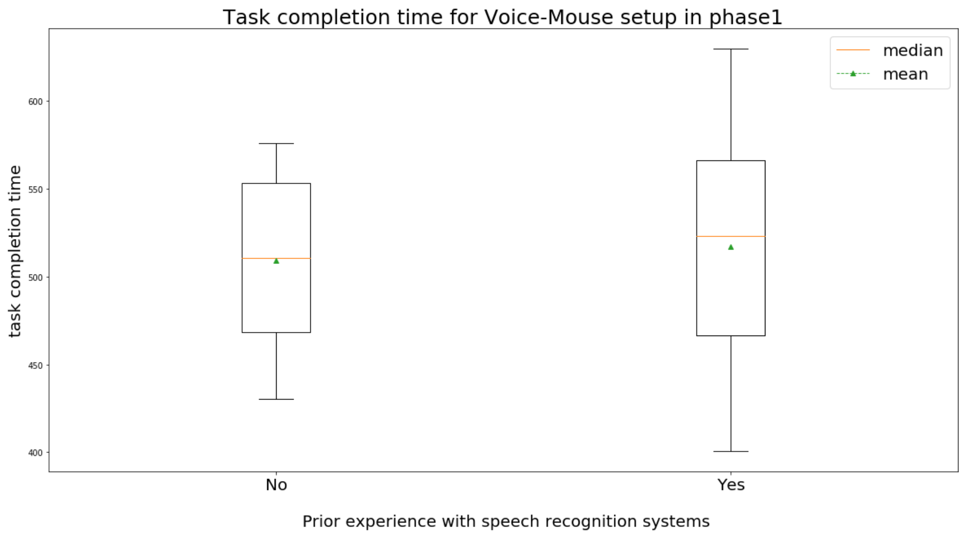


Figure 52: Time comparison for phase1 between participants with and without prior experience with speech recognition systems.

In the next step, we evaluated the effect of participants' keyboard shortcut expertise on Mouse-Keyboard time performances. Figure 53 shows the result. The Shapiro-Wilk test shows that the results are not drawn from a normal distribution. Therefore, we utilized Levene's Test instead of ANOVA. Levene"s test shows (p-value = 0.92) no difference between participants' time performance cross levels. This observation indicates that participants' level of expertise with keyboard shortcuts had no significant effect on their time performances.

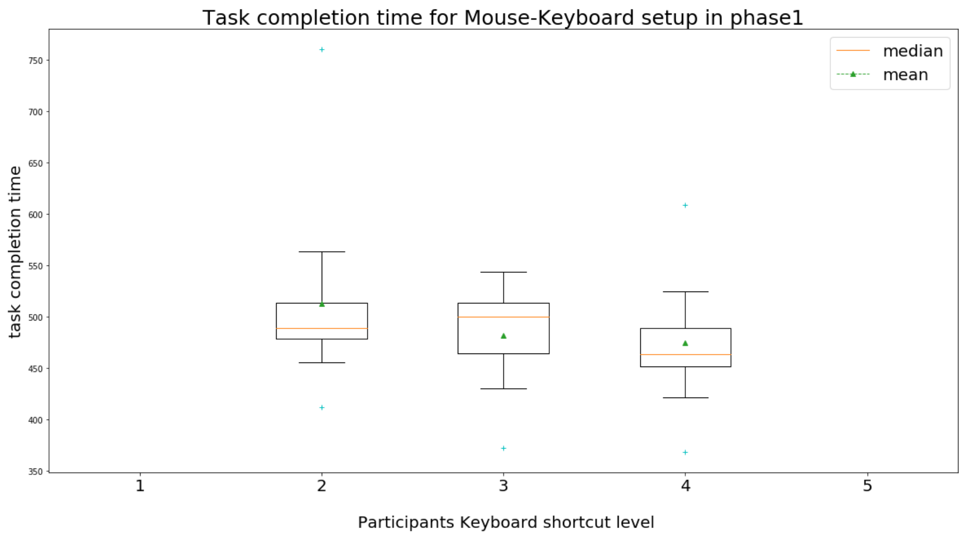


Figure 53: Time comparison for phase1 between participants level of expertise in using keyboard shortcut. Level 1 is the lowest and level 5 is the highest.

Finally, we evaluated the effect of participants' level of expertise in python on task completion time for each setup. Figure 54 shows the evaluation results. Python expertise level effect on Mouse-Keyboard results is not drawn from a normal distribution based on the Shapiro-Wilk test. Levene's test shows no significant difference (p-value = 0.23) in Mouse-Keyboard setups within expertise levels. Gaze-Keyboard and Voice-Mouse results are drawn from normal distributions. ANOVA test on Gaze-Keyboard ($F_{2,27}$ = 0.73, p-value = 0.48) and Voice-Mouse ($F_{2.27}$ = 1.32, p-value = 0.28) shows no significant differences. Therefore, we can conclude that participants' level of expertise in python did not significantly affect time performances.

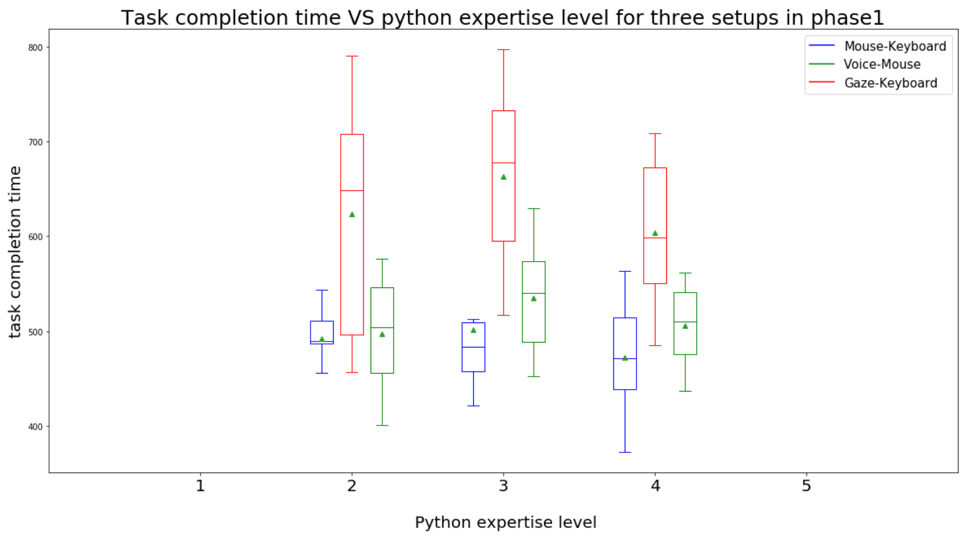


Figure 54: Time comparison for phase1 between participants level of expertise in Python programming. Level 1 is the lowest and level 5 is the highest.

6.3.4. Phase2 Results

The goal of phase2 of the experiment is to evaluate setups based on participants' formative feedback, as described in section 5.6 of this study. Although we performed formative evaluations in phase1, the results might not be convincing because of the experiment's controlled nature. In phase2, we use task4, task5, and task6, as mentioned in section 5.5 of this thesis. We give participants ten minutes for each task. We defined a scenario based on a real-world code maintainer where participants have to understand the code by reading it. Participants are free to perform any navigational action(s) they desire. This way, we give freedom to participants to test our setups without restriction. After the experiment, participants answered questionnaires regarding System Usability, Perceived Performance, NASA-TLX task load, and a short interview. Summative evaluations in terms of time is not suitable for this experiment phase. The reason is that the commands which participants perform are not identical across all of them. Therefore, the time comparison would not be informative. The task completion time is also irrelevant in this phase since it is a fixed time (10 minutes).

Accuracy: We evaluated the number of mistakes across all setups in this study. It is important to note that we cannot count all types of mistakes in this phase. The reason is that participants are free to perform any commands, and we do not know their intention. For example, if a participant wants to check the "variable last modified" and uses "variable initial" action instead, we cannot detect it in our logs. Besides, we cannot use the number of mistakes for each setup in this phase. The latter is because the commands set are different within participants and setups. Therefore, we use a mistake rate where we normalize mistake count by dividing them by commands count.

mistake rate = number of mistake commands / total number of commands

Figure 55 shows the results. The Shapiro-Wilk test shows that the results are drawn from a normal distribution. The ANOVA test shows that the result is statistically significant ($F_{2,27}$ = 19.54, p-value = 0.000006), indicating that the setup affects participants' accuracy. Mouse-Keyboard (μ = 0.08, M = 0.07) setup has the highest accuracy. Voice-Mouse was the least accurate (μ = 0.305, M = 0.306). Gaze-Keyboard was close (μ = 0.16, M = 0.12) to Mouse-Keyboard in terms of accuracy. Post-hoc comparison with using Tukey HSD shows that Voice-Mouse accuracy is significantly different from Mouse-Keyboard and Gaze-Keyboard (p-value = 0.001). However, there is no significant difference between Mouse-Keyboard and Gaze-Keyboard accuracy (p-value = 0.076). This observation is in line with our findings in phase1 Accuracy evaluation.

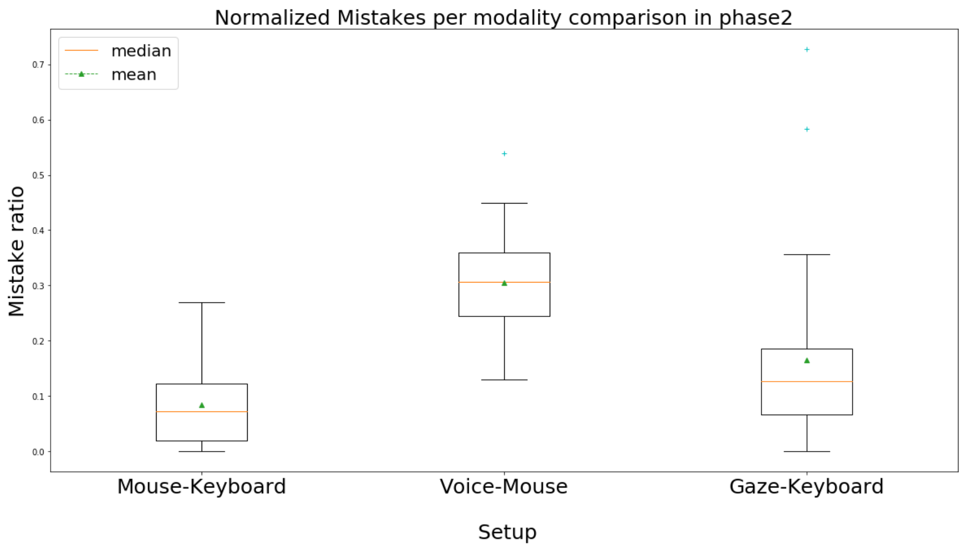


Figure 55: Mistake rate committed by participants in each setup for phase2 of the experiment.

Transcription Error Prediction: Figure 56 shows the prediction system results for the phase2. The success rate for predicting speech transcription error is 18.7 percent for phase2. It has an almost three percent drop compared to phase1. Step1 (voice pilot study results) was responsible for 10.8 percent of the success rate, and step2 (WordsAPI results) was responsible for 7.9 percent of the success rate. Step1 success rate has an 8.6 percent drop, and the Step2 success rate has a 5.1 rise compare to phase1 prediction results. The drop/raise in step1 and step2 success rate is because of the free nature of the phase2 experiment. In phase1, participants performed limited sets of commands, which were designed based on the voice pilot study. However, participants performed their desire commands in phase2 without restrictions, which lead to a drop in step1 success rate. However, step2 prediction covers the step1 to some extend. Therefore, we can conclude that, our approach for combining the vocal pilot study results and WordsAPI could reduce the transcription error rate successfully.

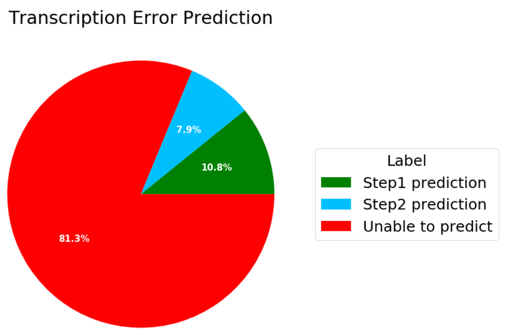


Figure 56: Transcription error prediction result in phase2.

System Usability Score (SUS): Table 13 shows the results for phase2 in detail. Although we observe some small drop/raise in phase2 SUS scores compare to phase1, the adjective rating, grade scale, and acceptability of setups did not change compared to phase1 results. We observe a higher standard deviation for Voice-Mouse and Gaze-Keyboard setups compare to Mouse-Keyboard. The reason is, for Voice-Mouse, we observed during the experiment that some participants faced higher transcription errors because of accent. For Gaze-Keyboard, participants with glasses faced issues during the calibration part. Also, some participants stated that using Gaze-Keyboard made their eyes a bit dry after the experiment.

Setup	**Score**	**Adjective**	**Grade**	**Acceptability**
Mouse-Keyboard	μ = 91.0 σ = 8.51	Excellent	A	Acceptable
Voice-Mouse	μ = 84.0 σ = 14.05	Good	B	Acceptable
Gaze-Keyboard	μ = 68.0 σ = 23.65	Ok	D	Acceptable (Marginal high)

Table 13: System Usability scores for setups in phase2.

NASA-TLX: NASA-TLX results (Figure 57) for phase2 were inline with phase1 results. Mouse-Keyboard (μ = 33.37, σ = 20.98) had the lowest physical-mental pressure, followed by Voice-Mouse (μ = 45.33, σ = 23.37) setup. Gaze-Keyboard (μ = 55.63, σ = 29.71) had the highest pressure compare to other setups. The primary pressure that some participants faced during Voice-Mouse setup were high transcription error. They told us that it was a bit cumbersome to repeat commands a couple of times. The primary sources of pressure for Gaze-Keyboard were difficult calibration because of glasses and dry eyes for some participants. Besides, some participants stated that they used to move their body and head a lot during code-related activities, making the Gaze-Keyboard setup challenging to use.

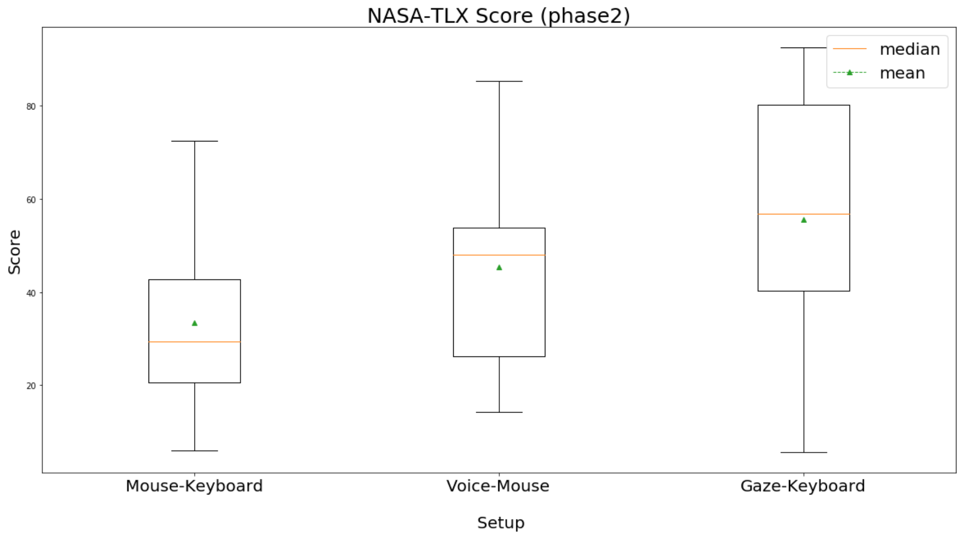


Figure 57: NASA-TLX score in phase2.

Perceived Performance: Figure 58 shows Perceived Performance results for phase2 experiment.

- Learnability: Voice-Mouse (μ = 5.7) and Mouse-Keyboard (μ = 5.6) obtained very a close score in this category. Gaze-Keyboard got a lower score (μ = 5.2) in this category. The Voice-Mouse and Mouse-Keyboard results are very close to phase1 results. For Gaze-Keyboard, we see a rise in the Learnability score. The latter is because participants already got familiar with the setup in phase1.
- Speed: Mouse-Keyboard (μ = 6.3) was the fastest setup in participants' opinions, followed by Voice-Mouse (μ = 5) setup. Gaze-Keyboard (μ = 3.6) obtained the lowest score. The reasons are similar to phase1, as participants stated. Voice-Mouse had transcription errors, which made participants repeat some commands. Besides, Slow scrolling and gaze deviation in Gaze-Keyboard slowed down participants. The results are in line with our finding in phase1 in terms of time performance.
- Accuracy: Mouse-Keyboard (μ = 6.0) was the most accurate setup in the participants' viewpoint. Voice-Mouse (μ = 5.0) had a 0.7 drop compare to the phase1. Gaze-Keyboard (μ = 3.9) was the least accurate, which had a 0.3 drop compare to phase1. Overall, the results are in line with the phase1. Participants gave a higher accuracy score to Voice-Mouse compare to Gaze-Keyboard despite the higher mistake rate of Voice-Mouse. As mentioned in phase1, the main reason was the participants' challenge with line selection due to the pupils' fluctuation and gaze deviation.

- Comfort: The most comfortable setup was Mouse-Keyboard (μ = 6.5) for participants. Voice-Mouse (μ = 5.8) and Gaze-Keyboard (μ = 4.0) were the next in terms of comfort. Transcription errors reduced participants' comfort in the Voice-Mouse setup. On the other hand, Gaze-Keyboard obtained the lowest score because of the physical barrier for participants. Some participants had dry eye issues after the experiment. Besides, some of them stated that it is difficult to sit in front of an eye tracker. The latter is because of the participants' habits to move his/her body and head frequently during code-related activities.

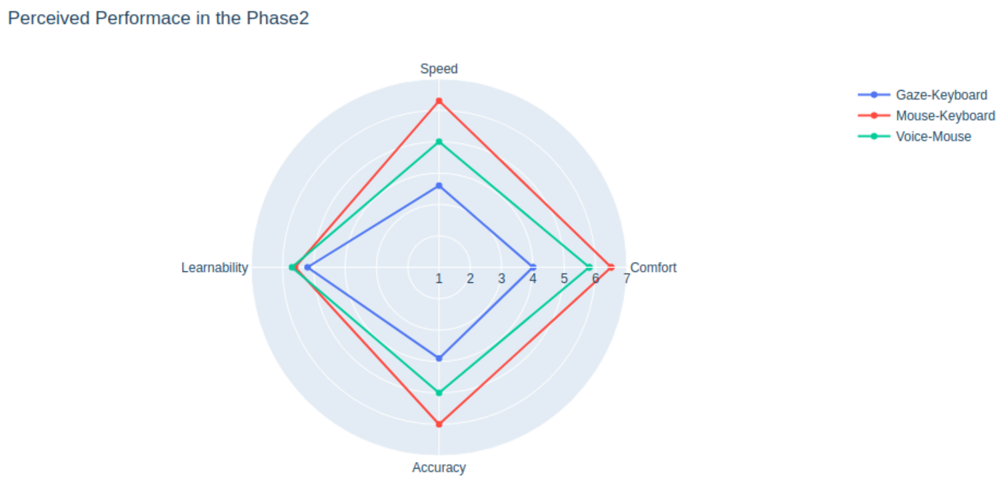


Figure 58: Participants Perceived Performance in phase2.

Participants' prior experience effect: We tried to evaluate the effect of participants' prior experiences obtained by phase0 (section 5.2) on phase2 formative evaluation results. First, we evaluated the effect of prior experience with eye-tracking on the Gaze-Keyboard System Usability Score. Figure 59 shows the result. There is no significant difference between the two groups ($t(5)$ = 0.18, p-value = 0.85) according to T-test. This result shows that participants' prior experience with eye-tracking did not significantly affect their opinion regarding the Gaze-Keyboard SUS score.

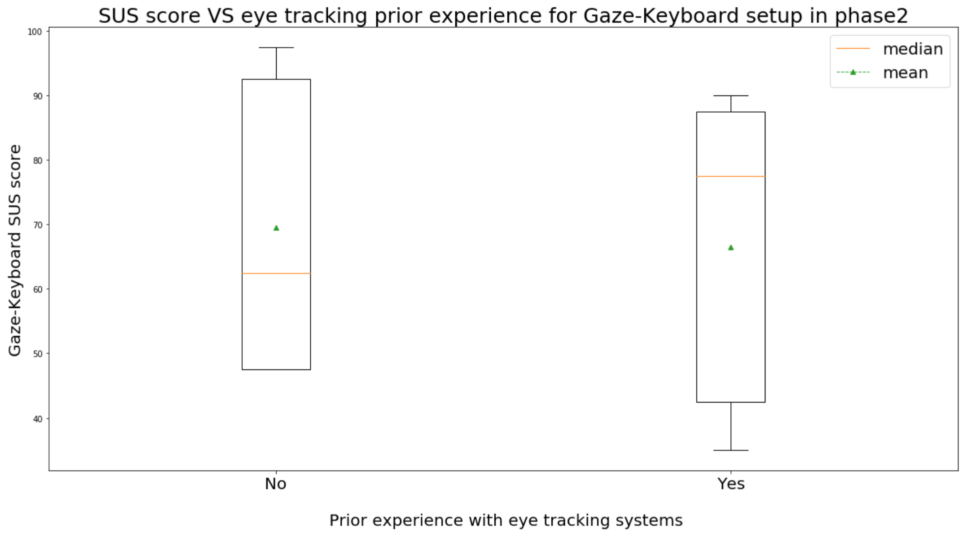


Figure 59: Evaluation of Gaze-Keyboard SUS score in phase2 based on participants prior experience with eye tracking.

Next, we tried to evaluate the effect of prior experience with speech recognition systems on Voice-Mouse SUS score. Figure 60 shows the results. There is no significant difference between the two groups ($t(6)$ = 0.82, p-value = 0.43) according to T-test. We can conclude that prior experience with speech recognition systems did not affect participants' opinions regarding Voice-Mouse SUS score.

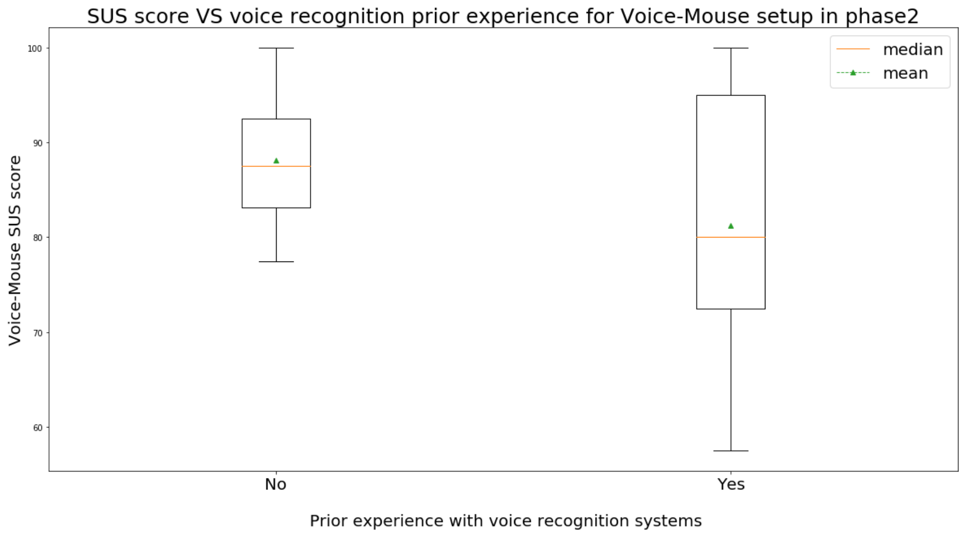


Figure 60: Evaluation of Voice-Mouse SUS score in phase2 based on participants prior experience with voice recognition.

Besides, we tried to evaluate the effect of participants' level of expertise in using keyboard shortcuts on their SUS score. In this evaluation, we considered all setups. The reason is we want to know the affect of traditional approach (keyboard shortcuts) on this study suggested setups. Figure 61 shows the results. ANOVA tests on results shows participants keyboard shortcuts level did not affect Mouse-Keyboard ($F_{2,7}$ = 0.15, p-value = 0.86), Voice-Mouse ($F_{2,7}$ = 0.22, p-value = 0.80), and Gaze-Keyboard ($F_{2,7}$ = 0.49, p-value = 0.62) SUS scores significantly.

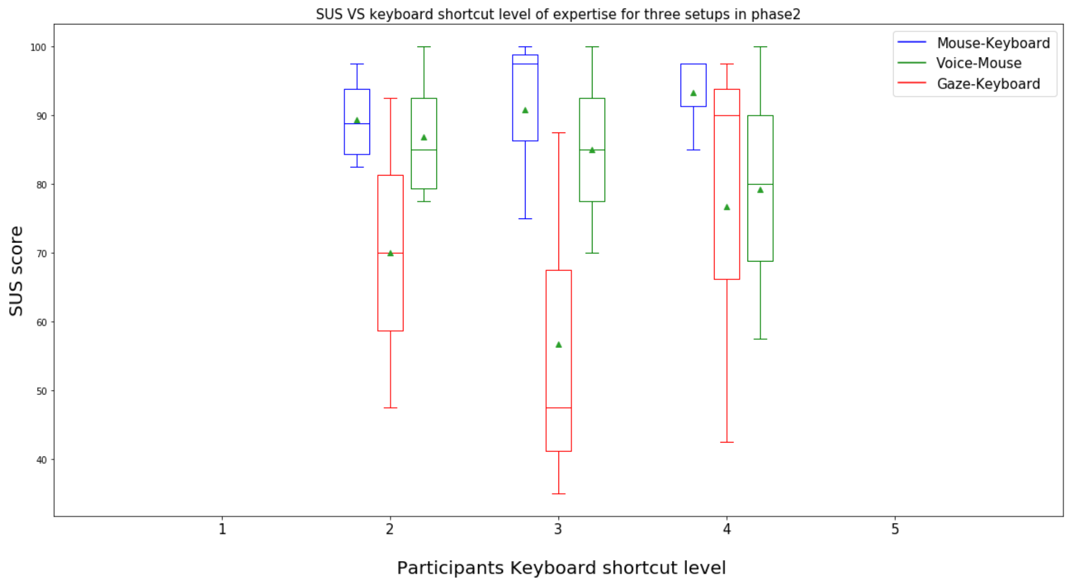


Figure 61: Evaluation of setups SUS score in phase2 based on participants prior experience with keyboard shortcuts.

Eventually, we tried to evaluate the effect of Python programming level of expertise on participants' opinions regarding the setups in this study. Figure 62 shows the results. ANOVA test shows that there is no significant difference in Mouse-Keyboard ($F_{2,7}$ = 3.49, p-value = 0.08) and Voice-Mouse ($F_{2,7}$ = 2.32, p-value = 0.16) results. However, the effect of participants Python level on Gaze-Keyboard ($F_{2,7}$ = 7.36, p-value = 0.01) SUS score is significant. To achieve better insight, we performed a Post-hoc comparison using Tukey HSD on each pair of levels for Gaze-keyboard SUS scores. The result shows a significant difference (p-value = 0.01) between level 2 and level 3 of the expertise's Python level. This observation means that participants with low Python expertise gave a lower SUS score to Gaze-Keyboard setups, as shown in Figure 62. One possible explanation for this effect is the combination of the Gaze-Keyboard pressure and phase2 experiment nature. We already showed that Gaze-Keyboard has higher task pressure than other setups in terms of NASA-TLX. On the other hand, the phase2 experiment demands participants to read the code and understand it. Consequently, participants have to deal with both the Python code and the new Gaze-Keyboard approach simultaneously. The combination of these two factors can affect the SUS score, as we see in Figure 62.

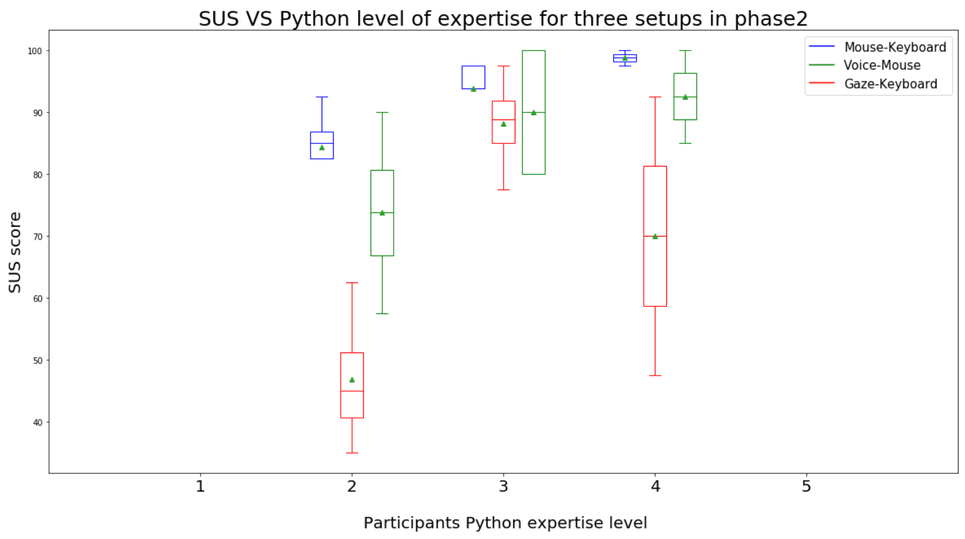


Figure 62: Evaluation of setups SUS score in phase2 based on participants prior experience with Python programming language.

7. Discussion

This chapter discusses the overall patterns and findings in our study to answer the research questions. We discuss the pilot studies followed by the design challenges and findings of each Multimodality approach in code navigation and, finally, the patterns and findings of this study based on the summative and formative evaluation results.

7.1. Pilot Studies

The survey results show that all navigational commands in this study were important in the programmers' viewpoint. The findings were of significance to us since it gave new insights into some of the new navigational actions that we introduced and evaluated, which were not present in the previous work. We can conclude that tracing a variable from initialization to the end and vice versa is crucial in code navigation, as suggested by the literature [7]. Online navigation actions (Search online, Next Result, Documentation) obtained higher importance than offline ones. The results also indicate that novice programmers might be more interested in online navigational actions (Table 11). These findings indicate the relevance of online navigational actions in coding. Therefore, there should be more studies regarding online navigation in terms of interaction since there is a gap in this field, as mentioned in the background literature (section 2).

Our vocal prediction system could reduce the transcription error rate. We think that with higher participation in our vocal pilot study, we could increase the success rate of valid predictions in the system. Other studies in this field that use the same commands can utilize our findings in Table 12 to reduce vocal commands transcription error rate. Our prediction approach is also useful in terms of vocal commands' target names (like identifiers in code). The latter is especially an advantage in the context of code-related vocal commands, where the domain of targets (except the Search command) is limited to the identifiers (variables, functions, APIs) in the programmer code. This advantage makes it possible to utilize similarity measures to predict a programmer's vocal input.

The vocal pilot study showed that we could personalize a vocal system for programmers by learning from their accent and common transcription errors. Besides, the survey shows we can have more natural vocal commands by utilizing the programmers' suggestions. These two findings show that we can personalize a voice-based IDE for programmers by providing the freedom to choose vocal commands instead of assigning a fixed command for all.

Results also indicate that "function" and "documentation" commands are more vulnerable to transcription errors than others. This indicates that choosing proper vocal commands is essential in a voice-based system. A proper command should be human language friendly and less vulnerable toward transcription errors.

7.2. Design Lessons

We faced some challenges during setup design and the main experiment. These challenges enabled us to understand some lessons regarding designing a Multi-modality approach for code navigation.

In the Mouse-Keyboard setup, the main challenge was to find the proper keyboard shortcuts. We had to add the Ctrl key to avoid possible collisions between our shortcuts and existing ones. This pushed us to choose three-keys shortcuts instead of two-keys. Having complex three-keys shortcuts is a potential drawback since three-keys shortcuts are more difficult to remember and use. For example, in our test experiment, we observed that the participant presses Ctrl+r, which reload the page instead of Ctrl+Alt+r for the Search online command. Overall, we can say that one potential drawback in the traditional mouse-keyboard approach is the complexity of adding more navigational actions. The reason is that adding more navigational actions demand designing new shortcuts.

Voiceye [22] utilized a mechanical switch to activate speech listening. However, we learned that the speech-to-text API's continuous listening is better than using an activation switch in our Voice-Mouse setup. We took this lesson from our vocal pilot study feedback. Some participants stated that they could not finish the vocal command after clicking the record button in the vocal pilot study application. The record button mimics the mechanical switch in Voiceye. The main challenge here is setting a proper waiting time to accept vocal input. This is a challenge since vocal commands have different lengths in our system. Some of them, like "Next," are very short. The others like "Search + query" are long. Therefore, we do not know how much precisely should the API waits for the input command. Thus, we decided to choose a continuous listening approach. Plus, using an activation key adds an extra interaction step while giving a vocal command, which can be a drawback.

We learned from our participants that having a "You said" box to see their vocal input transcription is necessary for Voice-Mouse setup. This is in line with Voiceye's findings, where their participants stated a similar concern. So all voice-based systems should support this feature. The other main challenge in Voice-Mouse was with API names. Most Python library functions have the "." character in their name (For example, numpy.divide). This makes them more vulnerable to transcription errors. We resolved this issue by mapping functions to their library so that the participants directly say the function name. Although this resolved the challenge in this study, it is not easy to implement this feature in real-world systems. Therefore, we can say that there is room to investigate the solution for overcoming high transcription error rates in API names.

Using the Ctrl key in Gaze-Keyboard for gaze clicking overcomes the Midas Touch issue in gaze-based systems. This is in line with Voiceye's findings, where they used a mechanical switch for gaze click. Our approach had one advantage compare to Voiceye since we used already existing keyboard modality instead of adding a new modality (mechanical switch). However, using a signal key for gaze clicking adds an extra step in gaze-based navigation. The extra step can add a time penalty to programmers' performance, as observed in Scrolling in the Gaze-Keyboard setup. Therefore, we can say there is a trade-off between accuracy and time in gaze-based systems.

We adopted the CodeGazer coloring approach to select targets for navigation in Gaze-Keyboard shortcuts. However, our Gaze-Keyboard setup is line-wise instead of CodeGazer region-wise methodology. This approach enabled our participants to jump to the target line and its identifiers directly. We added extra margins to each line since code lines are too close to each other to be selected by gaze. However, our participants stated that selecting a line was difficult since the gaze position fluctuated among close lines. Despite this issue, we can say that this approach can be better by future improvement in eye trackers' accuracy. The disadvantage of the coloring approach is that code line length has to be limited. The reason is we have seven colors, and therefore a line cannot have more than seven unique identifiers.

The last design lesson is regarding our Search navigational command in the Gaze-Keyboard setup. It is important to note that we think gaze typing like Voiceye is not a proper approach in code-related activities for non-disabled programmers. The reason is that programmers are often very skilled in typing with a keyboard. Therefore, gaze typing can slow them down. Therefore, gaze-based studies in code-related activity should focus on integrating the gaze modality into a keyboard or voice, which are faster typing ways. Participants faced a challenge during typing in the Gaze-Keyboard setup. Some of the participants used to gaze at the search box to check the query while typing. Sometimes, they gaze at the other parts of the monitor, which leads to moving the cursor out of the search box. Consequently, they faced multiple interruptions in typing.

7.3. Overall Results

Our results show that the Mouse-Keyboard setup obtained a favorable usability rate in both summative and formative evaluation areas. This is important because the Mouse-Keyboard setup was not a built-in Jupyter Notebook feature. Therefore, we can say that the Mouse-Keyboard setup fairly represents the traditional mouse-keyboard approach in code navigation.

In terms of time performance, Voice-Mouse setup results were comparable with Mouse-Keyboard. This result was in line with the Perceived Performance in terms of Speed where these two setups were close to each other. This shows that using a mouse for micro-interaction (like moving cursor) and using voice for additional more complex navigational commands can be a potential alternative for the traditional approach in terms of task completion time. Moreover, we can integrate voice as an add-on modality to traditional mouse-keyboard. For example, one can use voice for online navigational command and mouse-keyboard for others since the Voice-Mouse approach was especially faster than other setups in Search navigational command. Gaze-Keyboard was the slowest in both summative and formative measurements. Our line-wise selection approach empowered participants to have more natural code navigation with the gaze. However, some of the participants stated that the line selection issue was the main reason for feeling slow during code navigation. The line selection issue was because of line closeness to each other and small line sizes. This made it hard for some participants to gaze at a line because of eye tracker inaccuracy to pick small/close targets.

From an accuracy viewpoint, the Gaze-Keyboard approach was comparable with Mouse-Keyboard. This shows that adding a signal key (Ctrl key in our setup) can significantly overcome the Midas Touch issue in gaze-based systems. The Voice-Mouse was least accurate mainly because of transcription errors. However, participants' formative assessment shows that Voice-Mouse was more accurate than Gaze-Keyboard. There are three possible reasons for the contradiction between summative and formative results in terms of accuracy. Some participants stated that repeating a command in case of mistake was more cumbersome in Gaze-Keyboard than Voice-Mouse. Besides, some participants told us that having issues in line selection puts pressure on them. They had to focus on the target line to stop fluctuating gaze position between close lines. Although, in the end, they could select the line without mistake, it took too much effort to do it. The last reason was typing a search query in the Gaze-Keyboard setup, where they faced multiple interruptions in typing.

We performed a command-wise analysis to understand the proper additional modality for each command. Results showed that Voice is suitable for the Search command, where participants want to give a search query to the system. However, Voice did not perform well for Documentation command because of a higher rate of mistakes. This is because documentation names are often not human language friendly, which increases the possibility of transcription errors. On the other hand, Gaze was more proper for commands with no argument (Next, Go Back). The reason was that these types of commands demand no target selection. The command-wise analysis can help future systems to add additional modality based on their conventionality for each type of command. For example, a programmer can use Voice for search, Gaze for no argument commands, and the traditional approach for others.

In terms of modality NASA-TLX score (physical-mental pressure), Voice-Mouse was lower than the traditional approach. The reason was rooted in the transcription errors in the Voice-Mouse system. Participants stated that it was sometimes frustrating to repeat a command multiple times. Gaze-Keyboard obtained the highest pressure. As participants stated, the effort demanding line selection was a reason which put extra pressure on them. Besides, some of them told us that they had dry eye issues after the experiment. Moreover, some participants could not sit properly in front of the eye tracker since they often moved their bodies and heads while coding. The latter made Gaze-Keyboard a bit unnatural for them. We also showed that the setup pressure is especially important for participants with a lower level of Python programming expertise. This shows that similar studies should involve participants' programming level in their evaluation. A system might be acceptable for highly skilled programmers but not for novice ones. The reason is that novice programmers have to face the new modality pressure and programming pressure simultaneously.

Overall, the Voice-Mouse system was close to the traditional approach in formative and summative evaluations. It shows that voice is a natural modality for adding to the traditional approach for the more complex navigational command (like Search). The gap between Voice-Mouse and the traditional approach was related to the transcription error. We showed that adding a prediction system can reduce the errors in the Voice-Mouse system. Gaze-Keyboard had issues in line selection, interruption in a typing search query, and physical pressure on participants. However, it was marginally acceptable from the participants' viewpoint. For instance, the Gaze-Keyboard setup was suitable for commands with no argument. This shows that despite the issues, there is room for further improvement in the Gaze-Keyboard system.

8. Conclusion

Code navigation is an essential activity in programming, especially for debugging and maintenance tasks. Developers traditionally utilize mouse-keyboard and IDE features for code navigation. Studies show that traditional approaches are not efficient enough and add extra effort to developers' code-related activities. Therefore, researchers try to find new ways of interacting with source code. Unimodality and Multimodality approaches were suggested as alternatives for traditional approaches by using Voice and Gaze. However, alternative approaches have some limitations. Although they empower disabled programmers, their benefits for non-disabled programmers are not well studied. Besides, they do not support a wide range of navigational actions that raise questions regarding the system scalability. For instance, there is no work done on online navigation in terms of interaction. Moreover, their performance compares to the traditional approach are not clear. Additionally, both Gaze-based and Voice-based systems have some limitations, such as Midas Touch and transcription error.

In this thesis, we tried to find the natural way of integrating the additional modalities to the traditional mouse-keyboard approach. We also tried to evaluate alternative Multimodality approaches compare to the traditional approach. We added a broader range of code navigational actions compares to previous studies. For instance, we added online navigation for searching during code navigation. Our survey results show that all navigational actions in this study are important. For example, check a function definition is a crucial action from high skilled programmers viewpoint. Besides, online navigation seems to be an important action from novice developers' viewpoint.

To investigate the research questions, we implemented three Jupyter Notebook extensions for this research. To answer the first research question, we added Voice and Gaze as additional modalities to traditional approaches. Mouse-Keyboard extension represents the traditional approach in this study. The goal was to answer the second research question by comparing the traditional approach performance with alternatives. We implemented Voice-Mouse extension to add voice as an additional modality to the traditional mouse for code navigation. We established a vocal pilot study to decrease transcription errors in Voice-Mouse setup via a prediction system. The third extension was Gaze-Keyboard, where we added Gaze as an additional modality to the traditional keyboard. By adopting the CodeGazer coloring approach. However, with having line-wise gaze selection instead of region-wise. We also adopted Voiceye mechanical switch approach to overcome the Midas Touch issue.

We derived some design lessons for naturally adding a modality to traditional approaches based on our experiment results. The results show that the Voice-Mouse setup is comparable to the traditional approach in summative and formative evaluations. Our results show that Voice is an additional natural modality for having more complex navigational commands such as online searching. However, there is still a gap between Voice-Mouse and the traditional approach because of transcription error. We could successfully reduce the transcription error rate by the prediction system. This shows that we can overcome the transcription error issue by designing more efficient prediction systems in the future. Our additional Gaze modality adds extra pressure to participants compared to the traditional approach. The main reason was related to our line-wise selection, typing online search queries, and participants' lack of freedom in moving in front of the monitor. This extra pressure was particularly an issue for novice participants. However, Gaze-Keyboard was acceptable in participants' opinions. For instance, Gaze modality is especially useful for commands with no argument to select. This shows that Gaze can be a proper additional modality for traditional approaches with some improvements despite the issues.

9. Future Work

Our work has some limitations which can be addressed by future studies. Also, future studies can expand or improve some of the existing approaches presented in this study.

We presented some new navigational actions in this study, which are essential in code-related activities based on our survey. Future works can expand these navigational actions to investigate the scalability of Multimodality approaches. This is particularly vital in the context of online navigational actions where a gap in research exists. For instance, we can mix online navigational actions with recommendation systems to better experience both in cases of the relevancy of the search results and inter-action. Moreover, we showed that novice programmers might be more interested in online navigational actions. There should be more investigation on the relation between navigational actions' importance and programmers' expertise level.

We mentioned that designing new complex keyboard shortcuts for having more navigational actions is a possible drawback in the traditional mouse-keyboard approach. Future studies can investigate the traditional approaches' scalability by searching for ways to have easy-to-use keyboard shortcuts.

Transcription errors are the common disadvantage of all voice-based systems. Our prediction approach showed that we could reduce these errors by learning from the programmers' accent and common transcription errors. Therefore, future studies can search for ways to customize a voice-based system for programmers. Also, future studies can use more advanced approaches in machine learning and Natural Language Process to expand such prediction systems' ability. Moreover, we chose continuous listening over using an activation key for accepting vocal commands. We argued that using an activation key adds extra steps to interaction and can raise problems for long vocal commands based on our vocal pilot study experience. Future studies can investigate the advantages and disadvantages of using continuous listening compare to the activation key approach.

This study shows that we can overcome the Midas Touch issue in gaze clicking using an activation key on the keyboard. However, there might be a trade-off between time and accuracy. Future works can investigate the ways to eliminate this trade-off and have a high accurate/fast gaze clicking. Additionally, our Gaze-Keyboard approach has some issues with line-wise selection and Gaze-Keyboard typing, which can be improved in the future. Future works can also study how to expand the coloring approach for selection to support longer code lines with a high number of identifiers.

In phase2 of the experiment, we performed an uncontrolled experiment to evaluate approaches in a real-world scenario. For a better evaluation, future works can perform long-term evaluations by asking programmers to use these approaches on their daily programming and monitoring them over an extended time interval.

References

[1] About jupyter notebook. `https://jupyter.org/about`. Accessed: 2019-09-18.

[2] Jupyter-notebook extension. `https://jupyter-notebook.readthedocs.io/en/stable/extending/`. Accessed: 2019-12-11.

[3] Web speech api. `https://wicg.github.io/speech-api/`. Accessed: 2019-12-11.

[4] ABDI, H., AND WILLIAMS, L. J. Tukey's honestly significant difference (hsd) test. *Encyclopedia of research design 3* (2010), 583–585.

[5] AMANN, S., PROKSCH, S., NADI, S., AND MEZINI, M. A study of visual studio usage in practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (March 2016), vol. 1, pp. 124–134.

[6] BANGOR, A., KORTUM, P., AND MILLER, J. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies 4*, 3 (2009), 114–123.

[7] ERLIKH, L. Leveraging legacy system dollars for e-business. *IT professional 2*, 3 (2000), 17–23.

[8] FRITZ, T., SHEPHERD, D. C., KEVIC, K., SNIPES, W., AND BRÄUNLICH, C. Developers' code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 7–18.

[9] GLÜCKER, H., RAAB, F., ECHTLER, F., AND WOLFF, C. Eyede: Gaze-enhanced software development environments. In *Proceedings of the Extended Abstracts of the 32Nd Annual ACM Conference on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI EA '14, ACM, pp. 1555–1560.

[10] HART, S. G., AND STAVELAND, L. E. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, vol. 52. Elsevier, 1988, pp. 139–183.

[11] IQBAL, R., GRZYWACZEWSKI, A., HALLORAN, J., DOCTOR, F., AND IQBAL, K. Design implications for task-specific search utilities for retrieval and re-engineering of code. *Enterprise Information Systems 11*, 5 (2017), 738–757.

[12] KO, A. J., AUNG, H., AND MYERS, B. A. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th international conference on Software engineering* (2005), ACM, pp. 126–135.

[13] KRISTENSSON, P. O., AND VERTANEN, K. The potential of dwell-free eye-typing for fast assistive gaze communication. In *Proceedings of the symposium on eye tracking research and applications* (2012), pp. 241–244.

[14] LATOZA, T. D., VENOLIA, G., AND DELINE, R. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 492–501.

[15] LUTTEROTH, C., PENKAR, M., AND WEBER, G. Gaze vs. mouse: A fast and accurate gaze-only click alternative. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology* (New York, NY, USA, 2015), UIST '15, ACM, pp. 385–394.

[16] MAJARANTA, P., AHOLA, U.-K., AND ŠPAKOV, O. Fast gaze typing with an adjustable dwell time. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2009), pp. 357–360.

[17] MINELLI, R., MOCCI, A., AND LANZA, M. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension* (2015), IEEE, pp. 25–35.

[18] MINELLI, R., MOCCI, A., AND LANZA, M. Measuring navigation efficiency in the ide. In *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)* (2016), IEEE, pp. 1–6.

[19] MUDHOLKAR, G. S., SRIVASTAVA, D. K., AND THOMAS LIN, C. Some p-variate adaptations of the shapiro-wilk test of normality. *Communications in Statistics-Theory and Methods 24*, 4 (1995), 953–985.

[20] MURPHY, G. C., KERSTEN, M., AND FINDLATER, L. How are java software developers using the elipse ide? *IEEE software 23*, 4 (2006), 76–83.

[21] OHTA, T., MURAKAMI, H., IGAKI, H., HIGO, Y., AND KUSUMOTO, S. Source code reuse evaluation by using real/potential copy and paste. In *2015 IEEE 9th International Workshop on Software Clones (IWSC)* (March 2015), pp. 33–39.

[22] PAUDYAL, B., CREED, C., FRUTOS-PASCUAL, M., AND WILLIAMS, I. Voiceye: A multimodal inclusive development environment. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference* (2020), pp. 21–33.

[23] PIORKOWSKI, D. J., FLEMING, S. D., KWAN, I., BURNETT, M. M., SCAFFIDI, C., BELLAMY, R. K., AND JORDAHL, J. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2013), pp. 3063–3072.

[24] Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., and Lanza, M. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (New York, NY, USA, 2014), MSR 2014, ACM, pp. 102–111.

[25] Radevski, S., Hata, H., and Matsumoto, K. Eyenav: Gaze-based code navigation. In *Proceedings of the 9th Nordic Conference on Human-Computer Interaction* (New York, NY, USA, 2016), NordiCHI '16, ACM, pp. 89:1–89:4.

[26] Robillard, M., Walker, R., and Zimmermann, T. Recommendation systems for software engineering. *IEEE Software 27*, 4 (July 2010), 80–86.

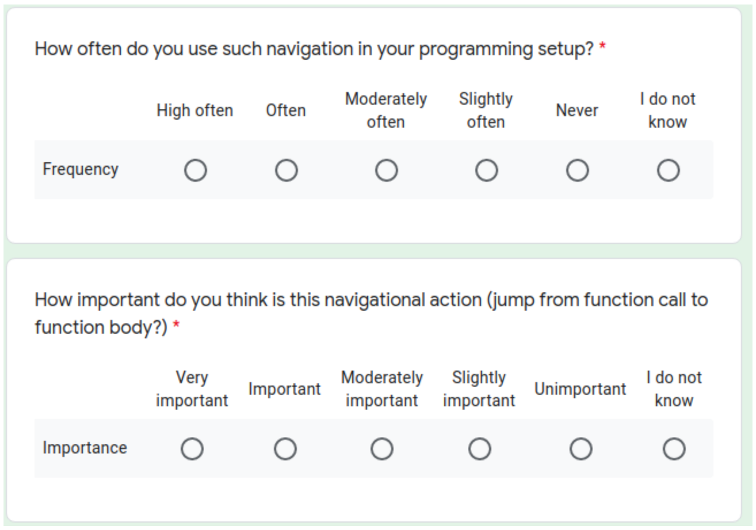
[27] Rosenblatt, L., Carrington, P., Hara, K., and Bigham, J. P. Vocal programming for people with upper-body motor impairments. In *Proceedings of the Internet of Accessible Things* (New York, NY, USA, 2018), W4A '18, ACM, pp. 30:1–30:10.

[28] Sadowski, C., Stolee, K. T., and Elbaum, S. How developers search for code: A case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 191–201.

[29] Shakil, A., Lutteroth, C., and Weber, G. Codegazer: Making code navigation easy and natural with gaze input. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2019), CHI '19, ACM, pp. 76:1–76:12.

Appendices

A. Survey Pilot Study parts



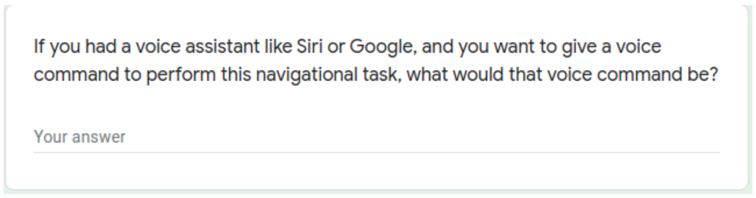
How often do you use such navigation in your programming setup? *

	High often	Often	Moderately often	Slightly often	Never	I do not know
Frequency	○	○	○	○	○	○

How important do you think is this navigational action (jump from function call to function body?) *

	Very important	Important	Moderately important	Slightly important	Unimportant	I do not know
Importance	○	○	○	○	○	○

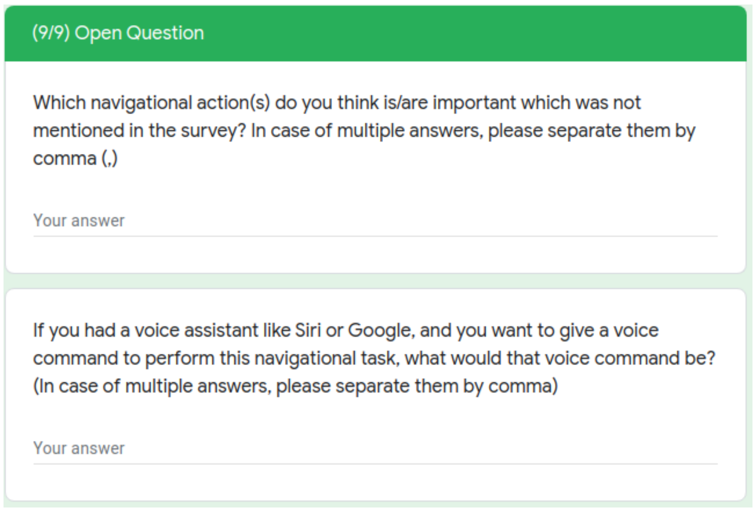
Figure 63: 6-points scale for importance and frequency of each navigational actions.



If you had a voice assistant like Siri or Google, and you want to give a voice command to perform this navigational task, what would that voice command be?

Your answer

Figure 64: Ask participants suggestion for choosing proper vocal command for each navigational action.



(9/9) Open Question

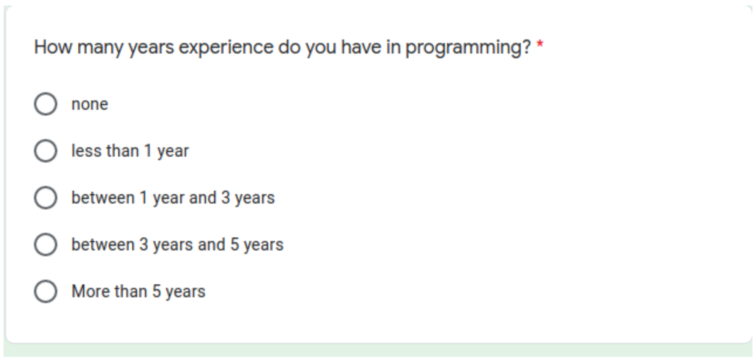
Which navigational action(s) do you think is/are important which was not mentioned in the survey? In case of multiple answers, please separate them by comma (,)

Your answer

If you had a voice assistant like Siri or Google, and you want to give a voice command to perform this navigational task, what would that command be? (In case of multiple answers, please separate them by comma)

Your answer

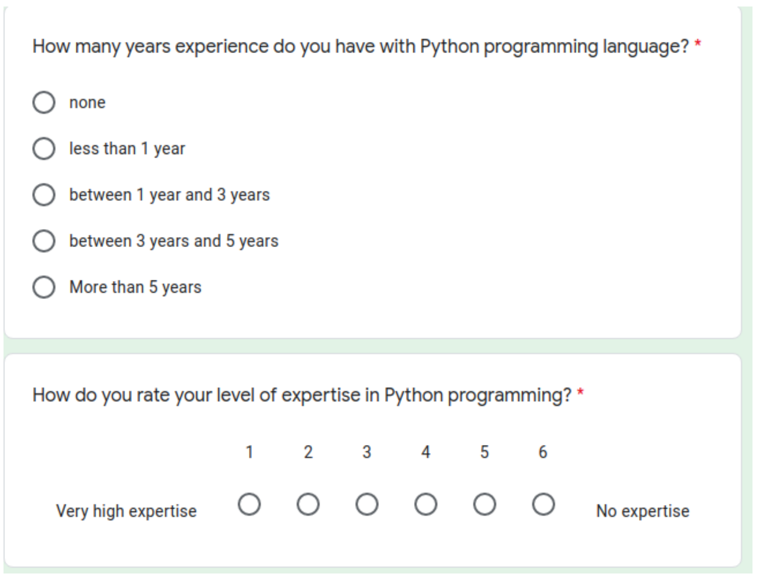
Figure 65: Ask participants opinion regarding other important navigational actions and their proper vocal commands.



How many years experience do you have in programming? *

- ○ none
- ○ less than 1 year
- ○ between 1 year and 3 years
- ○ between 3 years and 5 years
- ○ More than 5 years

Figure 66: Participants years of experience in programming.



How many years experience do you have with Python programming language? *

- ◯ none
- ◯ less than 1 year
- ◯ between 1 year and 3 years
- ◯ between 3 years and 5 years
- ◯ More than 5 years

How do you rate your level of expertise in Python programming? *

	1	2	3	4	5	6	
Very high expertise	◯	◯	◯	◯	◯	◯	No expertise

Figure 67: Participants years and level of expertise with Python programming. The level is from 1 (highest rank) to 6 (lowest rank).