

**SciPy 2024**

July 8 - July 14, 2024

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

Voice Computing with Python in Jupyter Notebooks

Blaine H. M. Mooers^{1,2,3}  

¹Department of Biochemistry and Physiology, College of Medicine, University of Oklahoma Health Sciences, Oklahoma City, OK 73104, USA, ²Laboratory of Biomolecular Structure and Function, University of Oklahoma Health Sciences, Oklahoma City, OK 73104, USA, ³Stephenson Cancer Center, University of Oklahoma Health Sciences, Oklahoma City, OK 73104, USA

Abstract

Jupyter is a popular platform for writing interactive computational narratives that contain computer code and its output interleaved with prose that describes the code and the output. It is possible to use one's voice to interact with Jupyter notebooks. This capability improves access to those with impaired use of their hands. Voice computing also increases the productivity of workers who are tired of typing and increases the productivity of those workers who speak faster than they can type. Voice computing can be divided into three activities: speech-to-text, speech-to-command, and speech-to-code. We will provide examples of the first two activities with the Voice-In Plus plugin for Google Chrome and Microsoft Edge. To support the editing of Markdown and code cells in Jupyter notebooks, we provide several libraries of voice commands at MooersLab on GitHub.

Keywords accessibility, productivity, human-computer interface

1. INTRODUCTION

Voice computing includes speech-to-text, speech-to-commands, and speech-to-code. These activities enable you to use your voice to generate prose, operate your computer, and write code. Using your voice can partially replace use of the keyboard when tired of typing, suffering from repetitive stress injuries, or both. With the Voice In Plus plugin for Google Chrome and Microsoft Edge, we could be productive within an hour. This plugin is easy to install, provides accurate dictation, and is easy to modify to correct wrong word insertions with text replacements.

We mapped spoken words to be replaced, called *voice triggers*, to equations set in LaTeX and to code snippets that span one to many lines. These *voice-triggered snippets* are analogous to traditional tab-triggered snippets supported by most text editors. (A tab trigger is a placeholder word replaced with the corresponding code when the tab key is pressed after entering the tab trigger. The existing extensions for code snippets in Jupyter do not support tab triggers.) We could use Voice In Plus to insert voice-triggered snippets into code and Markdown cells in Jupyter notebooks. Our voice-triggered snippets still require customizing to the problem at hand via some use of the keyboard, but their insertion by voice command saves time.

To facilitate voice commands in Jupyter notebooks, we have developed libraries of voice-triggered snippets for use in Markdown or code cells with the *Voice-In Plus* plugin. We are building on our experience with tab-triggered code snippets in text editors [1] and domain-specific code snippet libraries for Jupyter [2]. We have made libraries of these voice-triggered snippets for several of the popular modules of the scientific computing stack

Published Jul 10, 2024**Correspondence to**
Blaine H. M. Mooers
blaine-mooers@ouhsc.edu**Open Access** 

Copyright © 2024 Mooers. This is an open-access article distributed under the terms of the [Creative Commons Attribution 4.0 International](https://creativecommons.org/licenses/by/4.0/) license, which enables reusers to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the creator.

for Python. These voice-triggered snippets are another tool for software engineering that complements existing tools for enhancing productivity.

2. METHODS AND MATERIALS

2.1. Hardware

We used a 2018 15-inch MacBook Pro laptop computer. It had 32 gigabytes of RAM and one Radeon Pro 560X 4 GB GPU. We used the laptop's built-in microphone to record dictation while sitting or standing up to 20 feet (ca. 6 m) away from the computer.

2.2. Installation of Voice In Plus

We used the *Voice In* plugin provided by Dictanote Inc. First, we installed the *Voice In* plugin by navigating to the [Plugin In page](#) in the Google Chrome Web Store on the World Wide Web. Second, the [Microsoft Edge Addons web site](#) was accessed to install the plugin in Microsoft Edge.

We needed an internet connection to use *Voice In* because Dictanote tracks the websites visited and whether the plugin worked on those websites. *Voice In* uses the browser's built-in Speech-to-Text software to transcribe speech into text, so no remote servers are used for the transcription, so the transcription process was nearly instant and kept up with the dictation of multiple paragraphs. Dictanote does not store the audio or the transcripts.

After activating the plugin, we customized it by selecting a dictation language from a pull-down menu. We selected **English (United States)** from the 15 dialects of English. The English variants include dialects from Western Europe, Africa, South Asia, and Southeast Asia; many languages other than English are also supported.

Next, we set a keyboard shortcut to activate the plugin. We selected command-L on our Mac because this shortcut was not already in use. A list of Mac shortcuts can be found [here](#). This second customization was the last one that we could make for the free version of *Voice In*.

Customized text replacements are available in *Voice In Plus*. *Voice In Plus* is activated by purchasing a \$39 annual subscription through the **Subscription** sub-menu in the *Voice In Settings* sidebar of the *Voice In Options* web page. Monthly and lifetime subscription options are available. Only one subscription was required to use *Voice In Plus* in both web browsers. The library is synched between the browsers.

After purchasing a subscription, we accessed the **Custom Commands** in the *Voice In Settings* sidebar. We used the **Add Command** button to enter a voice trigger to start a new sentence when writing one sentence per line [Figure 1](#). The custom command for new paragraph can include a period followed by two new line commands, which works well when writing with blank lines between paragraphs and without indentation.

The phrases making up these voice triggers will no longer be available during dictation because they will be used to trigger text replacements. It is best to pick as voice triggers phrases that are unlikely to be spoken during dictation.

We kept the voice triggers as short as possible to ease accurate recall of the voice triggers. We had trouble correctly recalling voice triggers longer than four or five words. Three-word voice triggers are a good compromise between specificity and success at recall.

An exception to this guideline for shorter voice triggers was using two to three words at the beginning of a set of voice triggers to group them in the online *Voice In* library. Grouping related voice commands made finding them in the online library easier. For example, all Jupyter-related line magic voice triggers start with the phrase *line magic*. The prefix *line*

Edit Voice Command

Say This

To Insert This

☐ Match within a word

Edit

Figure 1. Entering a single voice trigger and the corresponding command in Voice In Plus.

1	line magic list magics	%lsmagic
2	line magic alias	%alias
3	line magic autocall	%autocall
4	line magic automagic	%automagic
5	line magic autosave	%autosave
6	line magic change directory	%cd

Figure 2. Snapshot of CSV file on Github for the [jupyter-voice-in](#) library.

magic is easy to remember, so it only adds a little to the recall problem. We show below a snapshot of the CSV file displayed in a pleasant format on GitHub for the Jupyter line magics [Figure 2](#). Note that these CSV files are atypical because they do not contain a line of column headers; *Voice In* does not recognize column headings.

The accuracy of the software's interpretation of the phrase was another limitation. We would replace frequently misinterpreted words with other words that were more often correctly interpreted.

We had to avoid making voice triggers that overlapped longer voice triggers to avoid unintended text replacements. For example, when we found a two-word phrase starting a four-word phrase, we would extend the two-word phrase to three words by selecting a third word to break the overlap between the two voice triggers. Otherwise, the two-word phrase would trigger a text replacement different from the text replacement expected from the four-word phrase.

We used the **Bulk Add** button to upload multiple commands from a two-column CSV file with commas as the field separator. We selected the file contents and pasted them in the open text box after clicking the **Bulk Add** button. The voice triggers reside in the left column, and the text replacements reside in the right column. The software ignored any capitalization in the voice trigger.

We handled multiline text replacements in one of two ways. First, we placed all the text on a single line and inserted the built-in command `<newline>` where linebreaks were needed. Second, we enclosed the multiline replacement text with one set of double quotes. Double quotes inside these text blocks had to be replaced with single quotes. We could use a backslash to escape internal pre-existing double quotation marks. Text replacements consisting

of commas also had to be enclosed with double else the commas would be misinterpreted as field separators. This also applied on Python code that contained commas.

We used the **Bulk Add** button to upload multiple commands from a two-column CSV file with commas as the field separator. We selected the file contents and pasted them in the open text box after clicking the **Bulk Add** button. The voice triggers reside in the left column, and the text replacements reside in the right column. The software ignored any capitalization in the voice trigger.

We handled multiline text replacements in one of two ways. First, we placed all the text on a single line and inserted the built-in command `<newline>` where linebreaks were needed. Second, we enclosed the multiline replacement text with one set of double quotes. Double quotes inside these text blocks had to be replaced with single quotes. We could not use a backslash to escape internal pre-existing double quotation marks.

The **Export** button opened a text box with the custom commands in CSV file format. We selected all the text box contents, copied them, and pasted them into a local CSV file using either the text editor TextMate or Emacs version 29.3. We used the **Synch** button to synchronize devices.

A GUI shows all the voice triggers and their text replacements immediately below the row of buttons mentioned above. Each row in the GUI has edit and delete icons. The edit icon opens a pop-up menu similar to the pop-up menu invoked by the **Add Command** button.

2.3. Construction of the snippet libraries

Some of our voice snippets had already been used for a year to compose prose using dictation. These snippets are in modular CSV files to ease their selective use. The contents of these files can be copied and pasted into the `bulk add` text area of the Voice In Plus configuration GUI.

2.4. Construction of interactive quizzes

We developed an interactive quiz to aid the mastery of the *Voice In Plus* syntax. We wrote the quiz as a Python script that can run interactively in the terminal or Jupyter notebooks. The quiz randomizes the order of the questions upon restart, ensuring a fresh experience every time. When you encounter a question you cannot answer, the quiz steps in to provide feedback, empowering you to learn from your mistakes and improve. Your wrongly answered questions are recycled during the current quiz session to allow you to answer the question correctly. This repetition helps build recall of the correct answers. We set a limit of 40 questions per quiz, allowing you to pace your learning and avoid exhaustion.

2.5. Availability of the libraries and quizzes

We tested the libraries using Jupyter Lab version 4.2 and Python 3.12 installed from MacPorts. All libraries are available at MooersLab on GitHub for download.

3. RESULTS

First, we describe the contents of the snippet libraries and the kinds of problems they solve. We group the libraries into several categories to simplify their explanation. Second, we describe the deployment of the snippet libraries for Voice In Plus (VIP).

3.1. Composition of the libraries

Our descriptions of these libraries include examples to illustrate how voice-triggered snippets work with automated speech recognition software. Developers can leverage our

libraries in two ways: by enhancing them with their unique commands or by using them as blueprints to create libraries from scratch.

The libraries are made available in a modular format, so the user can select the most valuable commands for their workflow. In general, our libraries are broad in scope, so they meet the needs of most users. Several libraries are domain-specific. These domain-specific libraries catalyze the creation of libraries tailored to other fields, sparking innovation and expanding the reach of voice-triggered snippets.

We divided the contents of the libraries into two categories. One subset of libraries supports dictating about science in the Markdown cells of Jupyter notebooks, while the other subsets support writing scientific Python in code cells. You can also apply these voice-triggered snippets to Markdown and code cells in Colab Notebooks. While some code, such as IPython line and cell magics, is specific to Jupyter, you can use most voice-triggered snippets to edit Markdown and Python files in Jupyter Lab.

Likewise, you can use these snippets in other browser-hosted text editors, such as the web version of Visual Studio Code because Voice In Plus works in most text areas of web browsers. You can use web services with ample text areas to draft documents and Python scripts with the help of voice-triggered snippets. You can also use Voice In Plus inside text boxes in local HTML files; however, Voice In Plus still requires an internet connection. You can edit the code or text with an advanced text editor using the GhostText plugin to connect a web-based text area to a text editor. Alternatively, you can save a draft document or script to a file and import it into Jupyter Lab for further editing off line.

3.2. *Libraries for Markdown cells*

These libraries contain a short phrase paired with its replacement: another phrase or a chunk of computer code. In analogy to a tab trigger in text editors, we call the first short phrase a voice trigger. Tab triggers are initiated by typing the tab trigger, followed by the tab key, which inserts the corresponding code snippet. Some text editors can autocomplete the tab-trigger name, so these text editors require two tab key entries. The first tab auto-completes the tab-trigger name. Then, the second tab leads to the insertion of the corresponding code. This two-step process of tab triggers empowers users with the flexibility to select a different tab trigger before inserting the code, enhancing the customization potential of text editors. It is important to note that voice triggers, while efficient, do not allow for the revision of the voice trigger. In the event of an incorrect code insertion, the undo command must be used, underscoring the need for caution when selecting voice triggers.

The most straightforward text replacements involved the replacement of English contractions with their expansions [Figure 3](#). Science writers do not use English contractions in formal writing for many reasons. Many automatic speech recognition software packages will default to using contractions because the software's audience is people who write informally for social media, where English contractions are acceptable. Adding the library that maps contractions to their expansions will insert the expansion whenever the contraction is spoken. This automated replacement of English contractions saves time during the editing process.

We grouped the voice triggers by the first word of the voice trigger [Table 1](#). This first word is often a verb. For instance, the word 'expand' is used before an acronym you want to replace with its expansion. Your memory of the words represented by the letters in the acronym often must be accurate. The library ensures that you use the correct expansion of the acronym. By providing instant access to the correct acronym expansions, the library significantly reduces the time that would otherwise be spent on manual lookups, allowing you to focus on your work. We have included acronyms widely used in Python programming,

Custom Commands

Enhance your experience with custom voice commands. These allow you to correct dictation errors, automate repetitive text entries, and perform actions like changing the dictation language, pressing keyboard shortcuts, etc.
For more information, [read our guide on using custom voice commands](#).







<div><div>+ Add Command</div><div>Bulk Add</div><div>Export</div><div>Sync</div></div>		
Say This	To Insert This	
'cause	because	 
'cuz	because	 
'twas	it was	 

Figure 3. Webpage of custom commands. The buttons are used to edit existing commands and add new commands. Three English contractions and their expansions are shown.

scientific computing, data science, statistics, machine learning, and Bayesian data analysis. These commands are found in domain-specific libraries, so users can select which voice triggers to add to their private collection of voice commands.

The customized commands are listed alphabetically in the Voice-In Plus GUI, with one command and its corresponding text replacement per row. The prefixes group like commands and, thereby, ease the manual lookup of the commands.

Some prefixes are two or more words long. For example, the compound prefix *insert Python* aids the grouping of voice triggers by programming language.

Another example of a verb starting a voice trigger is the command `display <equation name>`. This command is used in Markdown cells to insert equations in the display mode of LaTeX in Markdown cells. For instance, the voice trigger `display the electron density equation` is a testament to the convenience of our system, as shown in the transcript of a Zoom video [Figure 4](#). The image in the middle shows the text replacement as a LaTeX equation in the display mode. This image is followed by the resulting Markdown cell after rendering by running the cell.

Voice commands
expand acronyms
the book title
email inserts list of e-mail addresses (e.g., email bayesian study group)
insert Python (e.g., insert Python for horizontal bar plot)
insert Markdown (e.g., insert Markdown header 3)
list (e.g., list font sizes in beamer slides.)
open webpage (e.g., open google scholar)
display insert equation in display mode (e.g., display electron density equation)
display with terms above plus list of terms and their definitions (e.g., display electron density equation)
inline equation in-line (e.g., inline information entropy)
site insert corresponding citekey (e.g., site Jaynes 1957)

Table 1. Examples of voice commands with the prefix in bold that is used to group commands.

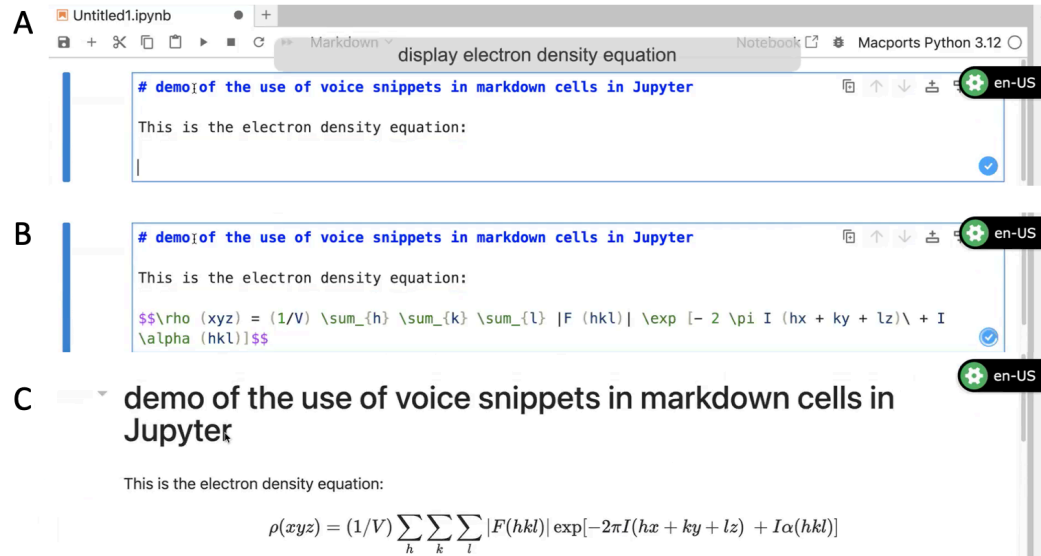


Figure 4. Three snapshots from a Zoom video of using the voice-trigger `display electron density equation` in a Markdown cell in a Jupyter notebook. A. The Zoom transcript showing the spoken voice trigger. B. The text replacement in the form of a math equation written in LaTeX in display mode in the Markdown cell. C. The rendered Markdown cell. The green and black tab on the right of each panel indicates that the Voice In plugin is active and listening for speech.

Likewise, the command `inline <equation name>` is used to insert equations in prose sections in Markdown cells. We have introduced voice-triggered snippet libraries for equations commonly used in machine learning and Bayesian data analysis <https://github.com/MooersLab/>. You can use these equations as templates to generate new equations. While the development of libraries is still in progress, they serve as flexible templates that you can adapt to your domain.

Some voice triggers start with a noun. For example, the voice trigger `URL` is used to insert URLs for essential websites.

Another example involves using the verb `list` in the voice trigger, as in `list matplotlib color codes`, to generate a list of the color codes used in Matplotlib plots. These voice triggers provide instant access to essential information, saving you the time and effort of manual searches.

The markup language code is inserted using the verb `insert`, followed by the markup language name and the name of the code. For example, the command `insert markdown itemized list` will insert five vertically aligned dashes to start an itemized list. The command `insert latex itemized list` will insert the corresponding code for an itemized list in LaTeX.

We have developed a library specifically for the flavor of [Markdown](#) utilized in Jupyter notebooks. This library is used to insert the appropriate Markdown code in Markdown cells. We have included a [library for LaTeX](#) because Jupyter Lab can edit tex files.

We have yet to figure out how to use voice commands to advance the cursor in a single step to sites where edits should be made, analogous to tab stops in conventional snippets. Instead, the built-in voice commands can move the cursor forward or backward and select replacement words. We included the markup associated with the `yasnip` snippet libraries to serve as a benchmark for users to recognize the sites that should be considered for modification to customize the snippet for their purpose.

3.3. Libraries for code cells

The `insert` command in code-cell libraries allows the insertion of chunks of Python code, enhancing your data science workflow. We avoid making voice commands for small code fragments that might fit on a single line. An exception to this was the inclusion of a collection of one-liners in the form of several kinds of comprehensions. As mentioned, we have developed practical chunks of code designed to perform specific functions, making your data analysis tasks more efficient. These chunks of code are functions that produce an output item, such as a table or plot. The idea was to fill a code cell with the code required to produce the desired output. Unfortunately, the user will still have to use the computer mouse to move the cursor back over the code chunk and customize portions of the code chunk as required.

While self-contained examples that utilize generated data can illustrate concepts, these examples are frustrating for beginners who need to read actual data and would like to apply the code example to their problem. Reading appropriately cleaned data is a common task in data science and a common barrier to applying Jupyter notebooks to scientific problems. Our data wrangling library provides code fragments that directly import various file types, easing the task of data import and allowing focus on downstream utilization and analysis.

After the data are verified as correctly imported, exploring them by plotting them to detect relationships between a model's parameters and the output is often necessary. Our focus on the versatile *matplotlib* library, which generates various plots, is designed to inspire creativity in data visualization and analysis [*matplotlib]. Our code fragments cover the most commonly used plots, such as scatter plots, bar graphs (including horizontal bar graphs), kernel density fitted distributions, heat Maps, pie charts, and contour plots. We include a variety of examples for formatting tick marks and axis labels as well as the keys and the form of the lines so users can use this information as templates to generate plots for their purposes. Generating plots with lines of different shapes, whether solid, dashed, dotted, or combinations thereof, is essential because plots generated with just color are vulnerable to having their information compromised when printed in grayscale. Although we provide some examples from higher-order plotting programs like *seaborn* [3], we focused on *matplotlib* because most other plotting programs, except the interactive plotting programs, are built on top of it.

We also support the import of external images. Images often play essential roles in the stories told with Jupyter notebooks.

3.4. Jupyter specific library

We provide a [library](#) of 85 cell and line magics that facilitate the Jupyter notebook's interaction with the rest of the operating system. Our cell magics, easily identifiable by their cell magic prefix, and line magics, with the straightforward line magic prefix, are designed to make the Jupyter notebook experience more intuitive. For example, the voice command *line magic run* inserts `%run`. You use this command to run a script file [Figure 5](#).

3.5. Interactive quiz

We developed a [quiz](#) to improve recall of the voice commands. These quizzes are interactive and can be run in the terminal or in Jupyter notebooks [Figure 5](#). The latter can store a record of your performance on a quiz.

To build long-term recall of the commands, you must take the quiz five or more times on alternate days, according to the principles of spaced repetition learning. These principles were developed by the German psychologist Hermann Ebbinghaus near the end of the 19th Century and popularized in English his 1913 book *Memory: A Contribution to Experimental*


```
[*]: %run -i 'qvoicein.py'
```

```
This quiz has 51 fill-in-the-blank or short-answer questions.
Each question in the quiz is asked just once if it is answered correctly.
Incorrectly answered questions will be recycled until they are answered correctly.
The questions are randomly shuffled upon start-up of the script, so each quiz is a new adventure!
If you do not know the answer, enter "return", and it will be printed to the terminal.

Say ____ to scroll up in the Chrome browser.  scroll up
Correct! :)

Say ____ to close tab in the Chrome browser.  open tab
The answer is "close tab".
Explanation: "1".
Find more information in 1.

Say ____ to insert an exclamation mark.  ! for history. Search history with c-1/c-1
```

Figure 5. An example of an interactive session with a quiz in a Jupyter notebook. The code for running the quiz was inserted into the code cell with the voice command *run voice in quiz*. The quiz covers a range of voice commands, including [specific voice commands covered in the quiz].

Psychology. They have been validated several times by other researchers, including those developed an algorithm for optimizing the spacing of the repetitions as function of the probability of recall [4]. Spaced repetition learning is one of the most firmly established results of research into human psychology.

Most people need discipline to carry out this kind of learning because they have to schedule the time to do the follow-up sessions. Instead, people will find it more convenient to take these quizzes several times in 20 minutes before they spend many hours utilizing the commands. If that use occurs on subsequent days, then the recall will be reinforced and retaking the quiz may not be necessary.

3.6. Limitations on using Voice In Plus

The plugin operates in text areas on thousands of web pages. These text areas include those of web-based email software and online sites that support distraction-free writing (e.g., [Write Honey](#)). These text areas also include the Markdown and code cells of Jupyter notebooks and other web-based computational notebooks such as Colab Notebooks. Voice In also works in plain text documents opened in Jupyter Lab for online writing.

Voice In Plus works in Jupyter Lite. It also works in streamlet-quill, which uses a Python script to generate a text box in the default web browser. It also works in the web-based version of [VS Code](#).

Voice In will not work in desktop applications that support the editing of Jupyter notebooks, such as the [JupyterLab Desktop](#) application, the [nteract](#) application, and external text editors, such as *VS Code*, that support the editing of Jupyter notebooks. Likewise, *Voice In Plus* will not work in Python Widgets. *Voice In Plus* is limited to web browsers, whereas other automated speech recognition software can also operate in the terminal and at the command prompt in GUI-driven applications.

Voice In Plus is very accurate, with a word error rate that is well below 10%. Like all other dictation software, the word error rate depends on the quality of the microphone used. *Voice-In Plus* can pick out words from among background ambient noise such as load ventilation systems, traffic, and outdoor bird songs.

The language model used by *Voice-In Plus* is quite robust in that dictation can be performed without an external microphone. We found no reduction in word error rate when using a high-quality Yeti external microphone. Our experience might be a reflection of our high-end hardware and may not transfer to low-end computers.

Because of the way *Voice-In Plus* is set up to utilize the Speech-to-Text feature of the Google API, there is not much of a latency issue. The spoken words' transcriptions occur nearly in real-time; there is only a minor lag. *Voice In Plus* will listen for words for 3 minutes before automatically shutting down. *Voice In Plus* can generally keep up with dictation occurring at a moderate pace for at least several paragraphs, whereas competing dictation software packages tend to quit after one paragraph. The program hallucinates when the dictation has occurred at high speed because the transcription has fallen behind. You have to pay attention to the progress of the transcription if you want all of your spoken words captured. If the transcription halts, it is best to deactivate the plugin, activate it, and resume the dictation.

Pronounce the first word of each sentence loudly so that they are recorded. Otherwise, the first words of your sentences will be skipped. This problem of omitted words is most acute when there has been a pause in the dictation.

The software does not automatically insert punctuation marks. You have to vocalize the name of the punctuation mark where it is required. You also have to utilize the built-in new-line command to start new lines. We have combined the `period` command with the `new line` command to create a new command with the voice trigger of `new sentence`.

You have to develop the habit of using this command if you like to write one sentence per line. This latter form of writing is helpful for first drafts because it eases the shuffling of sentences in a text editor during rewriting. This form of writing is also compatible with version control systems like git, which track changes by line number.

The practical limit on the number of commands is set by the trouble you will tolerate when scrolling up and down the list of commands. We had an easy time scrolling through a library of about 8,000 commands and a hard time with a library of about 19,000 commands. Bulk deletion of selected commands required assistance from the user support at Dictanote Inc. They removed our bloated library, and we used the **bulk add** button in the GUI to upload a smaller version of our library.

4. DISCUSSION

The following four discussion points can enhance understanding the use of *Voice-In Plus* with Jupyter.

4.1. *Independence from breaking changes in Jupyter*

The Jupyter project lacks built-in support for libraries of code snippets. Developing third-party extensions is not just a choice but a crucial necessity to support using code snippets in the Jupyter environment. Unfortunately, changes in the core of Jupyter occasionally break these third-party extensions. Users are burdened with the task of creating Python environments for older versions of Jupyter to work with their favorite outdated snippet extension, all the while missing out on the new features of Jupyter. An obvious solution to this problem would be for the Jupyter developers to incorporate one of the snippet extensions into the base distribution of Jupyter to ensure that at least one form of support for snippets is always available. Using voice-triggered snippets external to Jupyter side steps the disruption of snippet extensions by breaking changes in new versions of Jupyter.

4.2. *Voice-triggered snippets can complement AI-assisted voice computing*

The use of voice-triggered snippets requires knowledge of the code that you want to insert. The act of acquiring this knowledge is the up-front cost that the user pays to gain access to quickly inserted code snippets that work. In contrast, AI coding assistants can find code that you do not know about to solve the problem described in your prompt. From

personal experience, the retrieval of the correct code can take multiple iterations of refining the prompt. Expert prompt engineers will find the correct code in several minutes while beginners may take much longer. An area of future research is to use AI assistants that have large language models indexed on snippet libraries to retrieve the correct voice-triggered snippet.

4.3. Comparison with automated speech recognition extensions for Jupyter lab

We found three extensions developed for Jupyter Lab that enable speech recognition in Jupyter notebooks. The first, [jupyterlab-voice-control](#), supports custom commands and relies on the browser's language model. However, it is important to note that this extension is experimental and does not currently work with Jupyter 4.2. The second extension, [jupyter-voice-comments](#), relies on the DaVinci large language model to make comments in Markdown cells and request code fragments. This program requires clicking on a microphone icon frequently, which makes the user vulnerable to repetitive stress injuries. The third extension, [jupyter-voicepilot](#), is designed to provide a unique voice control experience. Although the extension's name suggests it uses GitHub's Copilot, it uses whisper-1 and ChatGPT3. This extension requires an API key for ChatGPT3. The robustness of our approach is that the *Voice-In Plus* should work in all browser-based versions of Jupyter Lab and Jupyter Notebook.

4.4. Coping with the imperfections of the language model

One of the most significant challenges in speech-to-text technology is the occurrence of persistent errors in transcription. These persistent errors may be due to the language model having difficulties interpreting your speech. For example, the language model often misinterprets the word *write* as *right*. Likewise, the letter *R* is frequently returned as *are* or *our*. It's crucial to have a remedy for these situations, which involves mapping the misinterpreted phrase to the intended phrase.

This remedy might be the best that can be done for those users who are from a country that is not represented by the selection of English dialects available in Voice In Plus. People from Eastern Europe, the Middle East, and Northeast Asia fall into this category. Users in this situation may have to add several hundred text replacements. As the customized library of text replacements grows, the frequency of wrong word insertions should decrease significantly, offering hope for improved accuracy in your speech-to-text transcriptions.

5. FUTURE DIRECTIONS

Our future directions include building out the libraries of voice-triggered snippets. Another direction includes the development of voice stops analogous to tab stops in code snippets for advanced text editors. The cursor would advance to the site of the next voice stop that you should consider editing to customize the code snippet for the project at hand. Another related advancement would be the concept of mirroring the parameter values at identical voice stops, similar to the functionality of mirrored tab stops in conventional snippets.

REFERENCES

- [1] B. H. Mooers and M. E. Brown, "Templates for writing PyMOL scripts," *Protein Science*, vol. 30, no. 1, pp. 262–269, 2021, doi: [10.1002/pro.3997](#).
- [2] B. H. Mooers, "A PyMOL snippet library for Jupyter to boost researcher productivity," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 47–53, 2021, doi: [10.1109/MCSE.2021.3059536](#).
- [3] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021, doi: [10.21105/joss.03021](#).

- [4] B. Tabibian, U. Upadhyay, A. De, A. Zarezade, B. Schölkopf, and M. Gomez-Rodriguez, “Enhancing human learning via spaced repetition optimization,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 10, pp. 3988–3993, 2019, doi: [10.1073/pnas.1815156116](https://doi.org/10.1073/pnas.1815156116).