Spring 2024

From Visual Data to Auditory Landscapes in ArcGIS Pro

Grace Hegland , '24

From Visual Data to Auditory Landscapes in ArcGIS Pro

Grace Hegland, Advised by Carr Everbach

Swarthmore College, Department of Engineering

May 10, 2024

Abstract

Data visualization is a central component to understanding spatial information in ArcGIS Pro. However, visualization creates an accessibility issue for visually impaired users. While ArcGIS supports two screen readers, JAWS and Microsoft Narrator, their ability to convey spatial relationships is limited to reading textual summaries and textual pop ups of spatial data [1]. This Senior Design project explores using data sonification to map ArcGIS data to piano pitches to create an auditory model of visually displayed map data. This auditory model allowed users to find facts and discover geospatial data trends. Ensuring fair access to information is essential for an equitable and inclusive workforce, and this project represents a significant stride towards enhancing the accessibility of ArcGIS Pro for the visually impaired.

Keywords: Data sonification, ArcGIS Pro, Inclusive design.

1 Introduction

1.1 Geographic Information Systems (GIS)

GIS is a system that allows users to visualize, create, analyze, manage, and map different types of data [2]. By integrating location data with descriptive attributes, GIS facilitates the interpretation of spatial patterns and relationships crucial for a myriad of projects [2]. ArcGIS Pro is a full-featured professional GIS desktop application created by Esri [3].

1.2 Spatial Data

The majority of spatial data is represented visually, which creates accessibility issues for the visually impaired. Individuals with visual impairments can interpret spatial data through textual summaries of graphs, but this does not allow for interaction with the data and cannot match all user needs [4]. One way to present spatial data to the visually impaired is through tactile methods. Tactile methods involve creating a raised, textured surface that users can feel to determine spatial relationships. Tactile methods, however, are cost-intensive and are typically only available in educational settings [4]. Consequently, to create a more accessible model of spatial data for the visually impaired, data sonification emerges as a promising technique.

Sound plays an important role in everyday experiences, as evidenced by the lack of sound in electric vehicles causing issues for drivers unaware that their vehicle has started [5]. Given the power of sound

and its presence in daily life, why is sound not more utilized in computers? Sonification overcomes this barrier and makes spatial data more accessible by turning data into non-speech based sound output [6]. The amount of spatial data is only increasing with projects such as Galileo, the European Union's global navigation satellite, and market penetration of GPS devices, making it more important than ever to make spatial data accessible [5].

1.3 Previous Work

Using Sound to Represent Spatial Data is a master's thesis from 2008 that created a sonification toolbar in ArcMap [5]. ArcMap is an older piece of software in Esri's collection of ArcGIS geoprocessing programs. ArcMap is in the process of being retired; it will continue to be supported until March 2026, but Esri recommends that users migrate their work from ArcMap to ArcGIS Pro [7]. The sound for this project was created using Musical Instrument Digital Interface (MIDI), which is the industry standard protocol for communicating musical events between computers, synthesizers, and digital instruments [5]. Specifically, MIDI piano pitches were used to convey changes in the data. The purpose of this project was to integrate sound and visual information display to avoid issues of visual saturation. Visual saturation occurs when adding additional data to a visual display would obscure the already existing data display [5]. Visual saturation can be avoided by representing specific data visually and conveying the remaining data through sonification. This sonification project was navigated by placing the mouse on a pixel, triggering its corresponding note.

Sonification of Climatic Data in MATLAB: An Acoustic Case Study is a Haverford senior thesis by Daniel Gillen from 2017 [6]. Gillen's project uses the mathematics of acoustic modeling, MATLAB, and trackpad interactions to create a sonified representation of climatic data [6]. Pitch was modeled using the fundamental frequency of sound and was increased as the data values increased [6]. To make sonified data accessible to the visually impaired, a navigation system different from the mouse placement used in *Using Sound to Represent Spatial Data* was required. The solution involved employing a Korg KP2 MIDI trackpad to navigate through the data [6].

Interactive Auditory Data Exploration: A Framework and Evaluation with Geo-referenced Data Sonification is an ACM Transactions on Computer-Human Interaction journal article from 2008 [4]. This paper explores the creation of iSonic, which is an interactive tool that allows users to find facts and discover trends of georeferenced data through sonification [4]. ISonic can be navigated with keyboard commands since most visually impaired people have memorized the keyboard layout [4]. The entire keyboard is mapped to the 2D screen position, and arrow keys can be used for relative movement [4]. ISonic maps to five pitches for sonification and the user can sweep through the map horizontally and vertically to construct a mental image of the data [4]. Additionally, the Ctrl+ command can be used to zoom into a range [4].

Now, the goal of my Senior Design project is to extend sonification functionality from ArcMap to ArcGIS Pro, focusing not only on reducing visual saturation but also on enhancing accessibility for the visually impaired. MIDI piano pitches are used to convey data trends as research has shown that musical sounds enhance numeric data comprehension [4]. Additionally, the program does not use the position of the mouse as the trigger for sound so that the technology can be accessible for the visually impaired. Instead, this project explored both trackpad interactions and a sweeping mechanism to allow the user to navigate through the sonified data in ArcGIS Pro.

2 Engineering Design

2.1 Constraints

There are several constraints that influence this project. The greatest constraint on this project is time. This project must be completed in a semester, which limits the amount of time to add new and improved features to the sonification. Other constraints on the project are the number of test subjects and the number of test sessions. To improve the accuracy of the assessments on the effectiveness and usability of the technology, it would be better to have more visually impaired individuals test the sonification. Additionally, it would have been ideal to have more test sessions and the time to implement all of the received feedback, creating an iterative design that best meets user needs.

2.2 Professional Standards and Codes

While there are standards for making content accessible, there are not standards for assessing the accessibility tools. For example, Web Content Accessibility Guidelines (WCAG) use success criteria to define three levels of conformance. However, these success criteria define ways of making data accessible through alternative media like sonification, but do not expressly define standards for the specific user requirements for these alternative medias. Additionally, *InfoSonics: Accessible Infographics for People who are Blind Using Sonification and Voice* suggests guidelines for sonification based on their results, advocating mapping the vertical axis to pitch and the horizontal axis to time, using musical sounds instead of pure sine waves, placing data between 50-75 milliseconds apart, using MIDI in the 35-100 range, and using stereo panning to separate data series [10]. While these guidelines are helpful metrics, they do not provide a standard that can be used to assess if a sonification project was a success. This paper draws its conclusion on the effectiveness of their infosonic product by comparing it to a text description and a sonification by asking the testers questions regarding engagement, aesthetics, understanding, and mental imagery [10]. However, there were no established standards in these categories for a successful product.

2.3 Requirements

Given the lack of clear standards for sonification, the design standards for this project are based on the percentage of questions users correctly answer and the principles of user-centered design (UCD). UCD emphasizes iterative development centered around user needs [11]. To prioritize UCD, users were asked to rank the ease of navigation of the technology on a scale of 1-10, targeting a rating of 5 or higher. Users were also asked a series of questions to assess understanding of two different maps. The model was considered a success if the users correctly answered 50% or more of the questions while rating the ease of the technology as a 5 or higher.

3 Methodology

3.1 Configuring the First Map

The project began with the selection of data to develop the sonification product from. Specifically, the ArcGIS raster data tutorial *Assess Hail Damage In Cornfields with Satellite Imagery* was chosen [12]. As shown in Figure 1, the starting data was satellite imagery before and after a hailstorm in Alberta, Canada. The satellite imagery after the hail storm is more washed out, but it is difficult to assess precise information about crop damage from these images [12].

Fig. 1: Satellite imagery of Alberta, Canada before (left image) and after (right image) a hail storm

Satellite sensors capture the amount of reflected light in different bands [12]. The spectral reflectance of a patch of healthy vegetation and patch of bare soil are shown in Figure 2. The difference in reflectance between the red light bands, shown in red, and the NIR light bands, shown in gray, is significantly wider for the healthy vegetation plot compared to the bare soil plot.

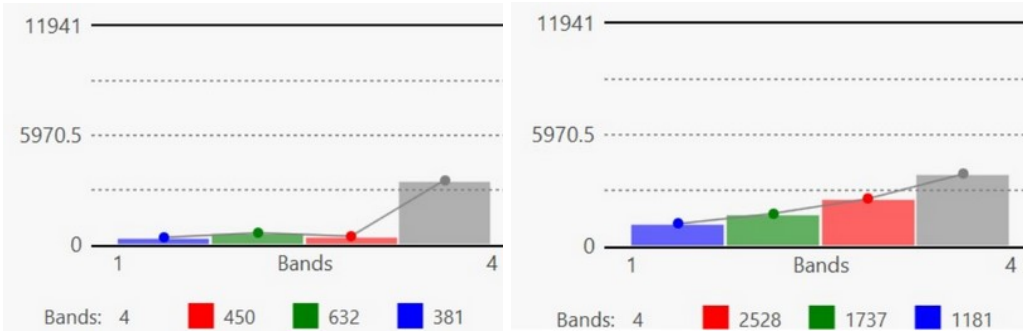

Fig. 2: Spectral reflectance of a patch of healthy vegetation (left image) and a patch of bare soil (right image)

The Soil-Adjusted Vegetation Index (SAVI) computes the gap between the red and NIR light reflectance

and gives a good measure of the health of vegetation [12]. ArcGIS Pro has a built-in tool for calculating SAVI values, which was used to produce layers for the SAVI values before the hail storm and after the hail storm. Figures 3 shows the resulting SAVI layers. The fields on the SAVI layer before the hail storm tend to be brighter, indicating higher SAVI index values and healthier vegetation [12]. The Compute Change raster function was then used to compute the difference between the two SAVI layers. The Remap raster function was also used to remove all the values below zero. The resulting loss of healthy vegetation layer is shown in Figure 4 with darker shades of purple indicating greater loss of healthy vegetation [12].

Fig. 3: SAVI before (left image) and after (right image) the hail storm

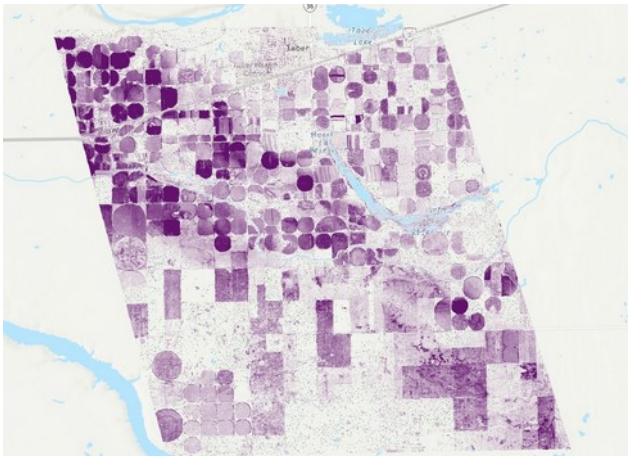

Fig. 4: Loss of healthy vegetation layer

Next, the Zonal Statistics as a Table geoprocessing tool was used to calculate the mean loss of healthy vegetation for each field. The resulting table was then joined to the layer for the field boundaries so that the symbology of the fields could be changed to show the range in mean crop damage values. The result is shown

in Figure 5 where red marks the fields with the most crop damage and yellow marks the fields with the least crop damage. The fields with the most damage tend to be along the diagonal from the northwest corner of the map to the southeast corner of the map, informing the user that the hail storm passed along this diagonal. It was decided that this map would work well for the sonification project because the sonification could allow visually impaired users to determine spatial patterns such as where the hail storm passed, where fields are more densely packed, and where there is the most crop damage.

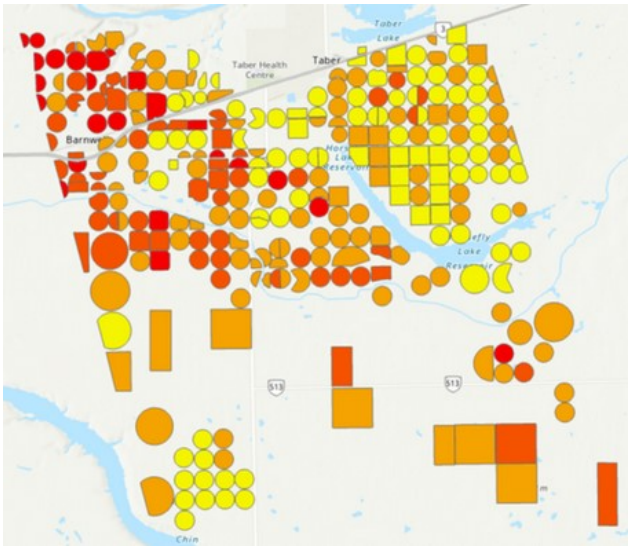

Fig. 5: Mean crop field damage after a hail storm in Alberta, Canada

3.2 Incorporating MIDI into ArcGIS

The next task was to incorporate MIDI into ArcGIS Pro. The ArcGIS Server comes bundled with a Python3 runtime. The ArcGIS Package Manager uses a custom Conda backend for managing the packages for its Python runtime. The python-rtmidi library was chosen to give the program access and bindings to MIDI functionality.

ArcGIS Pro provides a default Conda virtual environment. However, to add a Python library, one must create a new environment and do all of their modifications there. So the ArcGIS default environment was cloned and this clone was made the active environment. Then, python-rtmidi was pip installed via the ArcGIS Python command prompt into this new environment. The ArcGIS Python Command Prompt is a shell, initially in the default environment, with built in tools to manage packages and environments in ArcGIS Pro.

After pip installing python-rtmidi and subsequently running an ArcGIS Jupyter notebook, the rtmidi import failed. This issue was resolved by posing a question in the Esri community forum [13]. The response from the Esri community forum suggested iterating over each path in "sys.path" and printing the paths. The first path was the installation folder, the second path was the clone folder, and the sixth path was where the Python package should have been installed for the ArcGIS cloned environment to have access to it. Then,

the ArcGIS Python command prompt was checked to see where the python-rtmidi library was installed, and the installation files were copied over to the correct directory. After this step, rtmidi imported successfully.

3.3 Mapping Crop Field Damage to MIDI Notes

The next step was to determine how to map the mean crop field damage values to MIDI note numbers, ensuring that pitch rises as crop damage increases. The pitch mapping for this project draws inspiration from the methodology employed in a Python Sonification project that maps lunar craters over time to MIDI notes [14]. The first step was to export the field attribute table in ArcGIS Pro as a CSV file. Then, the pandas library was used to load the CSV file data as a pandas dataframe in an ArcGIS Pro Jupyter notebook [14]. The next step was to plot the data to examine the overall pattern and structure of the data [14]. The matplotlib library was used to create a scatter plot of mean crop field damage versus field ID as shown in Figure 6.

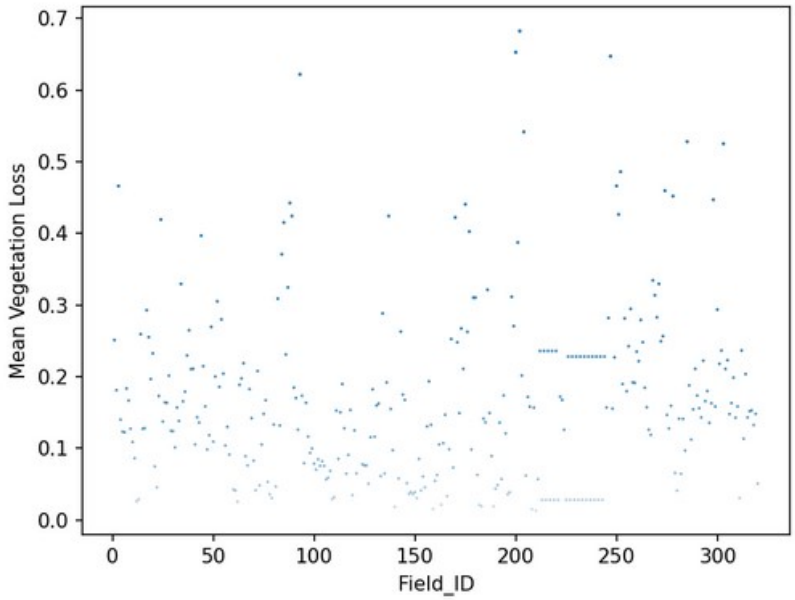

Fig. 6: Scatter plot showing mean vegetation loss by field ID

Then, a general mapping function was written as shown in Listing 1 [14]. This function takes an input value that lies in the range of a minimum value and a maximum value. Then, the function maps the value into a new range defined by a minimum result and a maximum result [14].

```
1 def map_value(value, min_value, max_value, min_result, max_result):
2     return min_result + (value - min_value)/(max_value - min_value)*(max_result - min_result)
```

Listing 1: General Mapping Function

The next step was to normalize and scale the data. To normalize the data, the general mapping function was utilized to map the range of means to lie between 0 and 1. Then, the data was plotted as a scatterplot using the matplotlib library as shown in Figure 7. The scatterplot indicates a broad range of mean crop field damage values, with the majority of fields exhibiting values below 0.4. Below a normalized mean vegetation damage of 0.4, many fields exhibited clustering around the same level of crop damage. Based on the scatterplot, it was decided to keep the scaling linear so that the fields clustered together are mapped to the same note, which allows the user to determine which fields had similar amounts of damage and to identify which fields had more substantial crop damage. A linear scaling allows for more grouping of the data and will help the user to pick up patterns such as the direction that the hail storm passed.

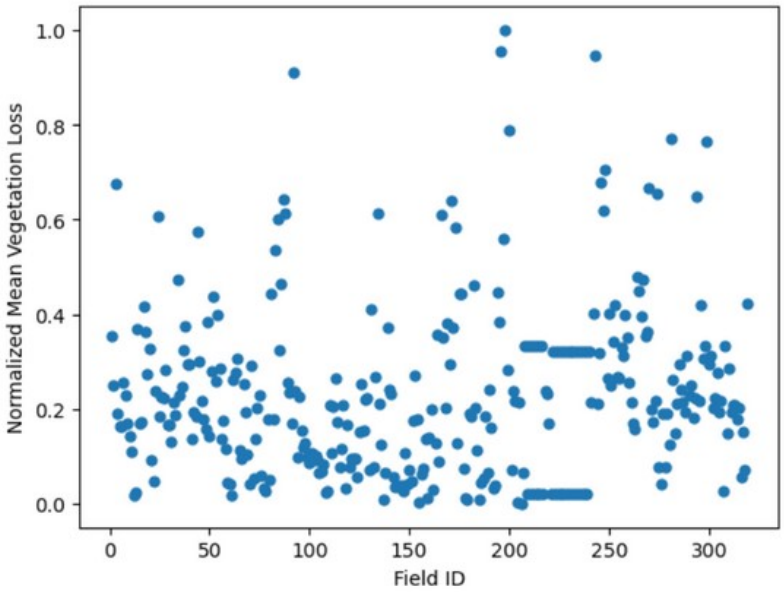

Fig. 7: Normalized and scaled mean vegetation damage by field ID

The next step was to determine the MIDI note numbers for the sonification. The major pentatonic scale was selected for this project because it does not have dissonant note combinations, which allows for a better user experience in the case that notes are played simultaneously [14]. The audiolazy library was used to find the corresponding MIDI note numbers for a couple of octaves in the major pentatonic scale [14].

The following step was to map the data to the MIDI note numbers as shown in Listing 2. An index for notes was generated using a for loop, employing the map value function to transform the normalized crop field damage data, spanning from 0 to 1, into a range determined by the MIDI note numbers [14]. The note index was then used to assign MIDI note numbers.

Then, the python-rtmidi documentation was referenced to determine how to play through the newly created MIDI data [15]. Next, the notes were played through by field ID, creating an auditory model of the scatterplot shown in Figure 7. The next step would involve transitioning from the scatter plot audio representation to an audio representation of the map to allow for geospatial understanding.

```
# Normalize data
y_data = map_value(dataToMap, min(dataToMap), max(dataToMap), 0, 1)

# MIDI note numbers for the major pentatonic scale
note_midis = [36, 38, 40, 43, 45, 48, 50, 52, 55, 57, 60, 62, 64, 67, 69, 72, 74, 76, 79, 81]

midi_data = []
    for i in range(numberOfRecords):
        note_index = round(map_value(y_data[i],0,1,0,19))
        midi_data.append(note_midis[note_index])
```

Listing 2: Mapping the data to MIDI note numbers

3.4 Testing Trackpad Navigation

The next portion of the project was to create a navigation system for moving through the map and listening to the notes. The original plan for the navigation system was to use a 2002 Korg KP2 Kaoss MIDI Trackpad, inspired by *Sonification for Climatic Data in MATLAB: an Acoustic Case Study.* To connect the trackpad to the computer, a driver was installed. After the driver installation, when available MIDI ports were printed, the trackpad showed up. Then, python code was used to read the x and y position from the trackpad. The x position was tracked accurately, but the y position was inaccurately tracked. Additionally, it was realized that this technology is outdated and no longer sold by the manufacturer. While the KP2 trackpad can still be found on eBay, it was decided that if we were trying to make spatial data from ArcGIS Pro more accessible to the visually impaired, it would be best not to require an already outdated piece of equipment that will only get more difficult to find.

3.5 Developing a Sweeping Navigation System

The next idea for a navigation system was inspired by the Isonic tool from *Interactive Auditory Data Exploration: A Framework and Evaluation with Geo-referenced Data Sonification.* Isonic uses a sweeping mechanism with keyboard commands for relative movement and zooming functionality. The first iteration of the sweeping code used the pyautogui library to drag the cursor horizontally. Then, a beep was added at the end of the line with the winsound library. Following the beep, the cursor's vertical coordinate would be decreased, moving the cursor down the page for its next sweep. The next step of this plan was to write a function to check if the cursor was hovering over a field during its sweep. However, there were challenges with creating this function, so instead, the sweep was modified to simulate a mouse click at every map pixel. For this idea to work, each pixel must know what field contains it in order to play the corresponding note. A dictionary that maps pixel values to fields based on field shapes and areas was considered as a possible solution. However, many of the fields are irregularly shaped, which would make implementing this solution challenging. After discussing these ideas with project advisor Carr Everbach, Carr suggested sweeping through the attribute table instead of the map.

Upon examining the attribute table, it was noted that the field ID numbers lacked a structured grid arrangement, thus failing to accurately represent the spatial layout of the fields. Therefore, it was decided that latitude and longitude must be added to the attribute table to sort the table spatially and generate

sweeps that accurately represent spatial relationships. Since latitude longitude coordinates can only be assigned to points, the ArcGIS Feature to Point tool was used to calculate and create a point layer that contained the centroids of all of the fields as shown in Figure 8. Then, the fields XLong and YLat were added to the attribute table, and the Calculate Geometry tool was used to populate these columns with the x and y coordinates from the centroids point layer. The Calculate Geometry tool utilized the GCS North American 1983 coordinate system, with the XLong and YLat data types specified as doubles.

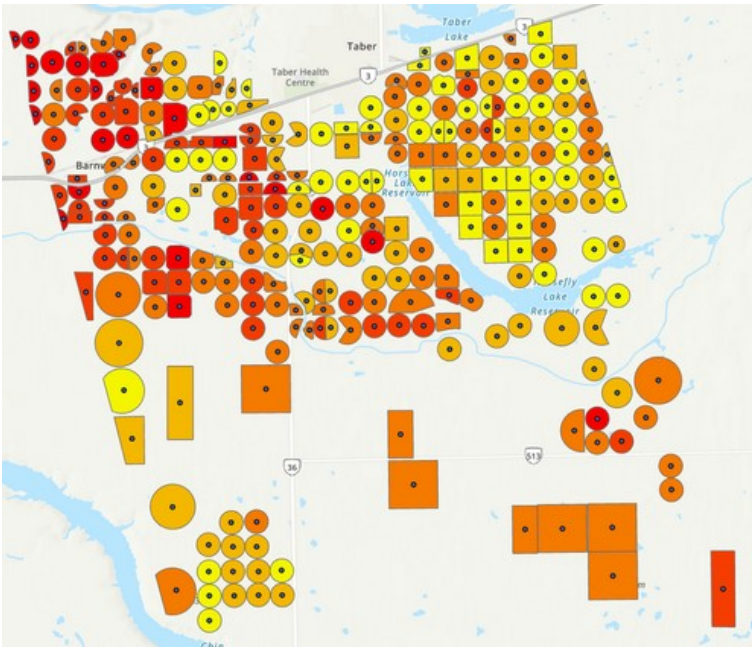

Fig. 8: Field layer and created centroid point layer

Then, a custom sort was applied to the attribute table to sort first by decreasing latitude and then by increasing longitude. After comparing the sorted attribute table to the map, it was realized that a tool more powerful than ArcGIS's custom sort would need to be built. Specifically, it was observed that latitudes within 0.003 of each other visually appeared as if they were on the same line in the map, but the custom sort placed them on separate lines. To address this issue, a dictionary was created that had latitude longitude pairs as keys and notes as values. Then, this dictionary was sorted first by decreasing latitude and then by increasing longitude, duplicating the results of the ArcGIS custom sort as shown in Listing 3. Next, a function was defined that filtered the dictionary into latitude bands with widths of 0.003 as shown in Listing 4. This filtered dictionary was then sorted by increasing longitude to produce the lines for the sweeps.

The next step was to integrate the filtered and sorted dictionary with the code to play MIDI notes. To complete this task, a while loop was created that started at the latitude corresponding to the top of the map and ran until it reached the latitude for the bottom of the map. Then, the MIDI code would play the notes from the filtered and sorted dictionary values. Additionally, a beep was added at the end of each line to inform the user when they had reached the end of the line. These beeps were also adjusted to decrease in frequency as the user moves South to assist with orientation.

```
# Create a dictionary with longitude and latitude pairs as the keys and notes as the values
latLongNotes = dict(zip(zip(data['XLong'], data['YLat']), midi_data))

# Sort the dictionary first by decreasing latitude and then by increasing longitude
sortedLatLongNotes = dict(sorted(latLongNotes.items(), key=lambda item: (-item[0][1], item[0][0])))
```

Listing 3: Creating a dictionary that duplicates the results of the ArcGIS custom sort

```
def in_range(latlong_and_note, bandWidth):
    latlong, _ = latlong_and_note
    long , lat = latlong

    return lat <= currentLat and lat > currentLat - bandWidth and long >= longrange[0] and long <= longrange[1]
```

Listing 4: The function used to filter sortedLatLongNotes

3.6 Adjusting Spacing and Orientation Along the Horizontal Axis

The next change that needed to be made was that the pause time between the notes needed to reflect the distance between the longitudes of the centroids of the fields. To accomplish this task, a function, delayCalc, was created for calculating delays. delayCalc takes the difference between two longitudes, divides this difference by the total interval width, and then multiplies by the total delay time for a line as shown in Listing 5. The total delay time for a line is determined by the number of notes in each line.

```
def delayCalc(long1, long2, delayTime, intervalWidth):
    return ((long2 - long1)/intervalWidth)*delayTime
```

Listing 5: Function that calculates the delay time between notes

A subsequent function, notesToDelays, uses the delayCalc function to calculate the delays between consecutive notes and then to append the calculated delays to a delays list as shown in Listing 6. Next, a function, midi_play, was created to play the MIDI note numbers as shown in Listing 7. A subsequent function, play_midi_notes, was defined to use the midi_play function to play the MIDI notes with time delays between consecutive notes as shown in Listing 8. An edgeDelays function was written, which calculates the delay time between the start of the line and the first note and the delay time between the last note and the end of the line as shown in Listing 9.

```
def notesToDelays(sortedLatLongNotes, delayTime, intervalWidth):
    delays = []
    longs = [long for long, _ in sortedLatLongNotes.keys()]
    for i in range(len(sortedLatLongNotes)-1):
        delays.append(delayCalc(longs[i], longs[i+1], delayTime, intervalWidth))
    return delays
```

Listing 6: Function to create a list of the delays between notes

```
def midi_play(note, noteDuration):
    note_on = [0x94, note, 127]
    note_off = [0x84, note, 0]
    midi_out.send_message(note_on)
    time.sleep(noteDuration)
    midi_out.send_message(note_off)
```

Listing 7: Function that plays MIDI notes

```
def play_midi_notes(sortedLatLongNotes, noteDuration, delayTime):
    timeDelays = notesToDelays(sortedLatLongNotes, delayTime, intervalWidth)
    for i in range(len(sortedLatLongNotes)):
        midi_play(list(sortedLatLongNotes.values())[i], noteDuration)
        if i < len(timeDelays):
            time.sleep(timeDelays[i])
```

Listing 8: Function that plays MIDI notes with the delays between them

```
def edgeDelays(longInitial, longFinal, delayTime, intervalWidth):
    initialDelay = ((longInitial - longitudes[0]) / intervalWidth) * delayTime
    finalDelay = ((longitudes[-1] - longFinal) / intervalWidth) * delayTime
    return (initialDelay, finalDelay)
```

Listing 9: Function for calculating the initial and final delay of a line

The next addition to the program was a continuous sine wave that decreases in pitch over a fixed time interval. This sine wave runs from the start of the line to the end of the line to help the orient the user along the horizontal axis while they hear piano notes. The sine wave was created using the pysinewave library as shown in Listing 10 [16].

```
def play_sine_wave(lineDuration):
    sinewave = SineWave(pitch=0, pitch_per_second= 5/lineDuration)
    sinewave.play()
    sinewave.set_pitch(-5)
    time.sleep(lineDuration)
    sinewave.stop()
```

Listing 10: Function for calculating the initial and final delay of a line

The functions were then incorporated into the while loop as shown in Listing 11. An if statement was added for if the note count in the line is greater than zero. It starts by calculating the note duration from the note time and the note count. Then, it uses the edgeDelays function to calculate the initial and final delay, and the initial delay starts. Threading is then used to simultaneously play the sine wave and the MIDI notes with the delays between consecutive notes [17]. The if statement ends with the final delay. There is a following else statement that is utilized if the note count of a line is zero. In this case, just the sinewave plays.

```
if noteCount > 0:
    sine_thread = Thread(target = play_sine_wave, args = (lineDuration,))
    sine_thread.start()

    noteDuration = noteTime / noteCount
    initialDelay, finalDelay = edgeDelays(longs[0], longs[noteCount-1], delayTime, intervalWidth)
    time.sleep(initialDelay)
    play_midi_notes(sortedLatitudeBands, noteDuration, delayTime)
    time.sleep(finalDelay)
else:
    play_sine_wave(lineDuration)
```

Listing 11: Code block contained in the while loop

3.7 Programming Keys for Navigation

The next step was to program the arrow keys for relative motion. The pynput library and documentation was used to write a function that would be the handler for a listener [18]. The handler function changed the current latitude value and beep frequency for the left, down, and up arrow keys and exited the program for the escape key. Therefore, at the end of each sweep of a line, the listener would listen for a left key to repeat the current line, a down key to move to the next line, and up key to move to the previous line, or the escape key to exit the program.

After adding line navigation, the "i" key was programmed to tell the user the current line. This was done by creating a variable called line counter whose value decreases when the down arrow key is pressed, increases when the up arrow key is pressed, and stays the same when the left arrow key is pressed. The Google Text-to Speech (gTTS) library was used to read the line counter as shown in Listing 12.

```
def say_line(line_counter):
    myText = f"line {line_counter}"
    language = 'en'
    output = gTTS(text = myText, lang = language, slow = False)
    output.save("output.mp3")
    os.system("start output.mp3")
```

Listing 12: The function that is called to read the line counter when the "i" key is pressed

Next, the "tab" key was programmed as the play key. Before the tab key was programmed in, every time the user pressed the down arrow, they would have to listen to a full line before moving on to a later line. With the tab key programmed as a play key, the user can press the down arrow, decide if they want to play that line, and press tab to play the line or the down arrow again to keep moving to later lines. To make this possible, the contents of the while loop were moved into the linePlay function as shown in Listing 13. Additionally, the while loop was replaced by a with block that contains just the listener as shown in Listing 14. This allows the listener to continue listening for key press combinations. The listener is stopped when the escape key is pressed or if the user navigates out of the latitude range. The implementation of the tab key rendered the left key press unnecessary, as users could now simply press tab after hearing a line to repeat it. Consequently, the detection of left key presses was removed from the handler.

```
def linePlay(currentLat, beep_frequency, lineDuration, noteTime):
    # Use the in_range function to filter the sorted note dictionary
    latitudeBands = dict(filter(lambda item: in_range(item, bandWidth), sortedLatLongNotes.items()))
    # Sort the latitude bands by increasing longitude
    sortedLatitudeBands = dict(sorted(latitudeBands.items(), key = lambda item : item[0]))
    noteCount = len(sortedLatitudeBands)
    longs = [long for long, _ in sortedLatitudeBands.keys()]
    delayTime = lineDuration - noteTime

    if noteCount > 0:
        # Use threading to simultaneously play the sine wave and the piano notes
        sine_thread = Thread(target = play_sine_wave, args = (lineDuration,))
        sine_thread.start()

        noteDuration = noteTime / noteCount
        initialDelay, finalDelay = edgeDelays(longs[0], longs[noteCount-1], delayTime, intervalWidth)
        time.sleep(initialDelay)
        play_midi_notes(sortedLatitudeBands, noteDuration, delayTime)
        time.sleep(finalDelay)
    else:
        # if there are no notes in the line, just play the sine wave
        play_sine_wave(lineDuration)

    winsound.Beep(beep_frequency, 500)
```

Listing 13: The function that uses the logic previously contained in the while loop

```
# Initialize all variables
currentLat = lats[0]
beep_frequency = 1000
line_counter = 1
longrange=[longitudes[0], longitudes[-1]]
zoom_counter = 0
intervalWidth = longitudes[-1] - longitudes [0]
bandWidth = (lats[0]-lats[-1])/60 # Fix the line number at 60 and calculate the bandwidth
lineDuration = 10
noteTime = 5

# Have the listener always listening with on_key_release as the handler
with midi_out:
    with Listener(on_release=on_key_release) as listener:
        listener.join()
```

Listing 14: The with block that contains the listener

The "s," "m," and "e" keys were programmed as line jumping keys that take the user to the starting, middle, and ending lines, respectively. This was done by setting the latitude, line counter, and beep frequency for these button presses as shown in Listing 15. This allows the user to jump between sections of the map with more ease, eliminating the need for repetitive pressing of the down arrow key to reach a specific section.

```
if key.char == "s":
    longrange=[longitudes[0], longitudes[-1]]
    currentLat = lats[0]
    beep_frequency = 1000 # Initial beep frequency
    line_counter = 1
    zoom_counter = 0
```

Listing 15: The "s" key programming that takes the user to the starting line

3.8 Programming Zoom Capabilities

The next feature added was zooming. This task was assigned to Madeline Mountcastle for her ENGR014 lab. It was explained to Madeline that her task was to program the "1" key to zoom into the left half of a region, the "2" key to zoom into the right half of a region, and the "0" key to return the zoom to the default. Therefore, if the "1" key was pressed two times, the program should zoom into the first fourth of the region. Madeline sent back her completed code, which adjusts the upper bound when the "1" key is pressed and adjusts the lower bound when the "2" key is pressed as shown in Listing 16.

After zooming capabilities were successfully integrated, the "z" key was programmed to read to the user the current zoom level. The zoom percentage was calculated using a zoom counter variable that gets incremented when a "1" or "2" key is pressed. The zoom level is read using the gTTS library. Next, the "h"

```
if key.char == "1":
    longrange[1] = longrange[0] + ((longrange[1] - longrange[0]) / 2) # Set a new upperbound
    zoom_counter += 1
```

Listing 16: The adjustment of the upper bound when the "1" key is pressed

key was programmed to function as a help key that reminds the user of the functions of each key. The help information is read using the gTTS library.

3.9 Touching Up the Code

After finishing with the programming of key press functionality, the next step was to edit the code to reduce the portions of the code that require hardcoded values. For example, the interval width was initially hard coded as 0.323. The interval width was rewritten by sorting the longitudes and then subtracting the minimum longitude from the maximum longitude. Additionally, the latitude band width was initially hard coded as 0.003, which gave the sonification 59 lines. To reduce the hard coding of the latitude band width, the total line number was fixed at 60 and a calculation using the minimum latitude, maximum latitude, and total line number was added to determine the latitude band width.

Additionally, the data originally loaded under the column name "MEAN," but the code was modified to instead take data from column 5 of the CSV. This modification allows for broader dataset compatibility. Also, initially, the code necessitated manual input of the CSV filename and its directory. Then, it was adapted into a command-line program, facilitating tab completion that avoids the tediousness of typing out both filename and directory.

3.10 Applying Code to Map of Swarthmore's Geoexchange System

Once the code was finalized for the first map, work on the second map began. Aegis Engineering project manager Beth Smith sent a DWG file detailing Swarthmore's geoexchange system. This data was opened in ArcGIS to produce the map shown on the left in Figure 9.

The data was initially all on one layer, so it was first separated into distinct layers. The attribute table for the polygon layer was queried to select the borehole polygons. Then, the Feature to Point geoprocessing tool was used to create a point layer for the centroids of the boreholes as shown on the right in Figure 9. The latitude and longitude for the centroids of the boreholes were then added as additional fields in the attribute table.

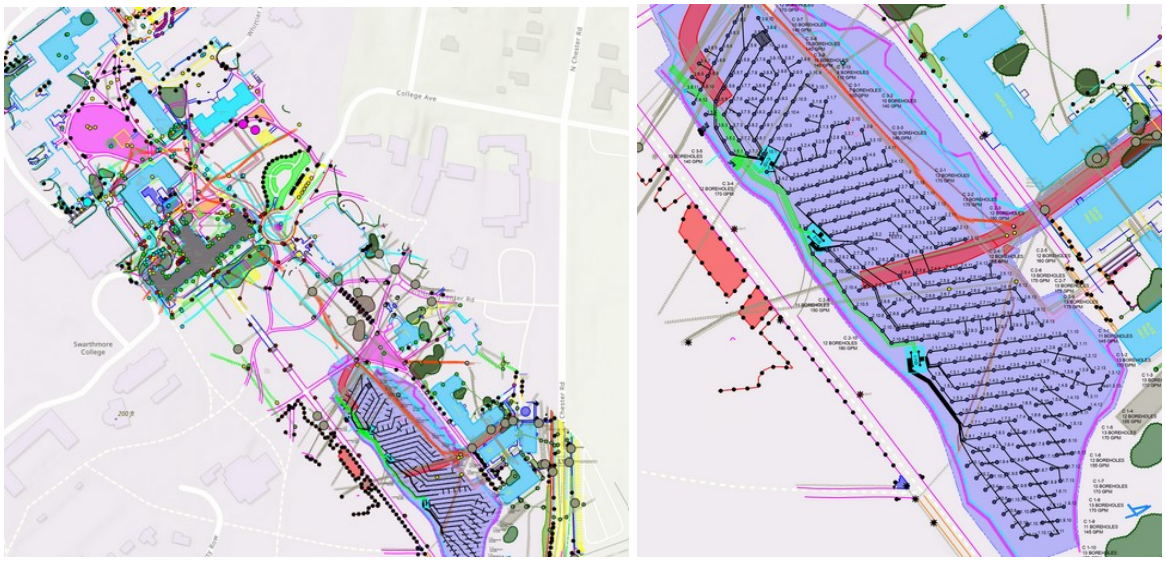

Fig. 9: Swarthmore's Geoexchange System: Left - Initial view, Right - After addition of the borehole centroids layer

The attribute table was exported as a CSV file and loaded into the code. Borehole depth data was allocated to column 5 for MIDI note mapping. However, since all of the boreholes have the same depth, data normalization mapped all the boreholes to not a number (NaN) values. To resolve this, an if statement checks if the minimum and maximum values of column 5 data are equal. If true, it assigns all data to note 57. Otherwise, the previously established procedure of normalizing the data and using a for loop to assign MIDI note numbers is followed.

After the NaN issue was resolved, testing was done to ensure that the longitude range, interval width, and line band width were being calculated correctly. Additionally, the sorted latitude bands were printed to check that spatial relationships were accurately being represented. All of the key press functionality was also tested and ran without issues.

3.11 Preliminary Testing

The next step was to create questions to access user understanding of both maps (Appendix A). The goal was to have questions that prompt users to identify specific facts from particular lines, as well as questions encouraging them to discern geospatial data trends across the entire map. After the questions were created, the next step was to undergo preliminary testing of the program.

Jalyn Miller and Madeline Mountcastle volunteered to test the program for their ENGR014 lab. While Jalyn and Madeline tested the program, it was observed that sometimes they would zoom into a region of a line, and then navigate to the next line before resetting the zoom. When they pressed tab to play the next line, they thought that they were listening to the full line when in actuality, they were still zoomed in. To address this issue, an addition was made to the code for the down arrow key, the up arrow key, the "s" key, the "m" key, and the "e" key that resets the zoom to the default level.

After completing the questions, Jalyn and Madeline were asked if there was anything they would change about the program. They asked for the ability to speed up the lines in the sonification. They thought that increasing the speed of the sonification would enable them to more effectively discern map-wide geospatial trends. Two more functions were written to address this need. The speed up function cuts the line duration and the note duration in half, and the slow down function increases the line duration and speed by a factor of two. Then, the left and right arrow keys were added to the handler so that if the right arrow key is selected, the speed up function runs, and if the left arrow key is selected, the slow down function runs.

3.12 Final Product Testing

The first final product tester was Hayden Dahmm. Hayden is a 2015 Swarthmore Engineering alum and McCabe Engineering award recipient. Hayden lost his sight before coming to Swarthmore, and while at Swarthmore, Hayden worked on various projects to make engineering more accessible for the visually impaired. For example, he worked with Ian Everbach to 3D print plastic click-wheels mounted on K'Nex sticks as tactile tools to represent electrical circuits that supported his studies in Swarthmore's ENGR011 course [19]. Additionally, he worked with Carr Everbach and 1993 Swarthmore alum Youngmoo Kim on the development of data sonification tools [20].

The second final product tester was Rose West. Rose is a 2026 Swarthmore political science major. Rose is also visually impaired, and while at Swarthmore, she has worked with staff and administration to improve disability programming on campus.

Testing with Hayden and Rose was conducted on separate days, but following the same methodology. On test day, users were first introduced to the tab key, the up and down arrow keys, and the zoom keys. Then, they were given some time to explore the map of mean crop field damage using these keys. The rest of the keys were then introduced, and users worked through some additional lines to test the functionality of the remaining keys. Then, users were asked the nine questions about the crop field damage map and the five questions about the geoexchange map to assess how well the program allowed them to interpret the data. Finally, the users were then asked to rate the usability of the program on a scale from 1 to 10.

4 Results

4.1 Results of Testing with Hayden Dahmm

Hayden answered 100% of the questions correctly. Additionally, on a scale from 1 to 10, Hayden rated the usability of the program an 8/10. For context, he mentioned that he would rate the Microsoft Edge Browser when it was first released a 0 and the JAWS screen reader a 9. Hayden mentioned that he especially appreciated the specialized and intuitive shortcut keys for navigating the audio map and the text to speech prompts for interrogating the data. Hayden also provided suggestions that could be incorporated into further program iterations to achieve a higher usability score.

First, Hayden highlighted the importance of incorporating a user introduction to the program, which would play notes at various crop damage levels. This feature would provide users with a clearer understanding of the significance of different pitches at the start of the sonification. This introduction could also include information about what numerical values each pitch corresponds to, allowing for better data comprehension at the start of the program. Furthermore, Hayden proposed enabling the option to switch the sweep direction from West to East to North to South, allowing for the detection of diverse data trends. Carr made the additional suggestion that the map could potentially be rotated and then swept.

The only question that Hayden said was difficult to answer was the geoexchange map question "on line 14, which of the two boreholes are the greatest distance apart?" Hayden mentioned that the zoom feature helped him answer this question, but that it would be easier to assess the delay length between the notes if the notes were more staccato. Hayden also mentioned that a nice addition to the crop field damage map would be to have the notes span the width of the fields. This adjustment would enable him to better comprehend the sizes of each damaged field. Hayden mentioned that it would be ideal to be able to switch between the note method of staccato notes and the note method where note length corresponds to object width.

Hayden also proposed ideas on enhancing data representation and fostering a broader understanding of the data. He suggested adding other instruments to communicate other data layers. Furthermore, to get a better big picture of the data, he suggested integrating the capability to simultaneously play multiple lines.

4.2 Results of Testing with Rose West

Rose answered 93% of the questions correctly and rated the technology's usability as a 8 or 9 out of 10. She offered two recommendations for improving program navigation. First, she proposed incorporating a command to halt playback midline and proceed to the next line. She highlighted situations where partway through the line, she realized that further listening was unnecessary and emphasized the convenience of being able to move on after having this realization. She also suggested having more keys like the "s," "m," and "e" keys to facilitate more frequent line jumping. Specifically, she mentioned that she would prefer smaller jumping increments, such as 15 instead of 30 lines, for more precise navigation.

5 Discussion

Hayden and Rose both answered well above 50% of the questions correctly and both rated the usability of the technology well above a 5, indicating that the project successfully met its design criteria. These results suggest that sweeping data sonification navigated through keyboard commands effectively facilitates the interpretation of ArcGIS data for individuals with visual impairments. Hayden explicitly mentioned that the keyboard commands eased data navigation beyond his initial expectations.

While this project was successful in meeting its objectives, there is more work that can be done to further improve the program. First, Hayden and Rose's suggestions should be incorporated into the code. Then, an important next step is to make the ArcGIS Pro interface more accessible for the visually impaired. Specifically, this project focused on map interpretation, but it is also important that the ArcGIS tools for geoprocessing, querying, and analyzing are accessible for the visually impaired. This would likely require testing how screen readers currently interact with ArcGIS Pro and writing Python code to aid the screen readers in providing effective navigation through the interface.

A limitation of this study was that testing was done on a computer that had a Swarthmore license for ArcGIS but did not have a JAWS or Microsoft Narrator set up. Therefore, Hayden and Rose were not able to assess the user friendliness of running the command line program in the ArcGIS Python Command Prompt. In upcoming phases of this project, it's recommended to add JAWS or a Microsoft Narrator to the computer for testing. With this addition, users could work with a map lacking pre-calculated latitude and longitude data. This setup would enable testing of the user's ability to navigate the ArcGIS Pro interface, extract the necessary data, export it as a CSV file, and then run it as a command line program in the ArcGIS Python Command Prompt.

In future iterations of this project it would be beneficial to increase the number of test participants.With a larger pool of participants, a better understanding can be drawn of the technology's effectiveness for making spatial data and ArcGIS Pro more accessible for the visually impaired. This expanded insight will aid in refining the interface and creating features to better cater to the diverse needs of users.

6 Conclusion

The purpose of this project was to create a Python program to allow visually impaired users to interpret spatial data in ArcGIS Pro. Both final tests surpassed the established design criteria, with users achieving correct responses well above the 50% mark and rating the program's usability considerably above a 5. These outcomes demonstrate that data sonification is a promising technique for conveying spatial information in ArcGIS Pro. Furthermore, the results suggest that a sweeping mechanism, keyboard navigation, and text to speech prompts created a navigable user interface. This project marks a significant step toward enhancing accessibility for the visually impaired within ArcGIS Pro. However, further efforts are necessary to fine-tune the user interface, guarantee accessibility of ArcGIS Pro's geoprocessing and analysis tools, and assess the user-friendliness of executing the command line program in the ArcGIS Python Command Prompt with the aid of a screen reader.

The methodology of this project also holds potential for broader applications. Specifically, the approach employed can be adapted to enhance the accessibility of scatter plots and CSV data in environments outside of ArcGIS Pro. This underscores the broader significance and versatility of the methodologies developed in this project, which have the potential to significantly improve accessibility for visually impaired users across various platforms.

Acknowledgements

The author acknowledges the Swarthmore College Department of Engineering for the support and academic space for this research. The author would like to specifically acknowledge Carr Everbach for advising and supporting this project throughout the semester. Additionally, the author would like to acknowledge Hadjer Ahner, Andrew Reuther, Alec Barreto, Doug Willen, and Madeline Mountcastle for their technical support. The author would also like to thank Hayden Dahmm, Rose West, Jalyn Miller, and Madeline Mountcastle for testing the program and providing invaluable feedback.

References

[1] Esri. "Accessibility in ArcGIS Pro." Esri, https://pro.arcgis.com/en/pro-app/latest/get-started/overview-of-accessibility.htm.

[2] Esri. "What is GIS?" Esri, https://www.esri.com/en-us/what-is-gis/overview.

[3] Esri. "Introduction to ArcGIS Pro." Esri, https://pro.arcgis.com/en/pro-app/latest/get-started/get-started.htm.

[4] Zhao, Haixia, et al. "Interactive Auditory Data Exploration: A Framework and Evaluation with Geo- referenced Data Sonification." ResearchGate, 1 Jan. 2006.

[5] Bearman, Nick. "Using Sound to Represent Spatial Data in ArcGIS." ResearchGate, 1 Dec. 2008, pp. 157-163, https://doi.org/10.1016/j.cageo.2011.12.001.

[6] Gillen, Daniel. "Sonification of Climatic Data in MATLAB: An Acoustic Case Study." Haverford College Department of Physics, 28 Apr. 2017.

[7] Esri Technical Support. "ArcMap Continued Support." Esri, 23 Jan. 2024, https://support.esri.com/en-us/knowledge-base/arcmap-continued-support-000029721.

[8] Swarthmore College. "To Zero By Thirty-Five." Swarthmore College, 2024, https://www.swarthmore.edu/to-zero-thirty-five.

[9] Fey, Olivia. "Energy Plan Student Engagement." Senior PSRF 2022-2023, 2023.

[10] Holloway, Leona M., et al. "Infosonics: Accessible Infographics for People who are Blind using Sonification and Voice." Association for Computing Machinery, April 2022, Article No. 480, pp. 1-13. DOI: 10.1145/3491102.3517465.

[11] Novoseltseva, Ekaterina. "User-Centered Design: An Introduction." Usability Geek, usabilitygeek.com/user-centered-design-introduction/.

[12] Viswambharan, Vinay, and Delphine Khanna. "Assess Hail Damage in Cornfields with Satellite Imagery." Esri, https://learn.arcgis.com/en/projects/assess-hail-damage-in-cornfields/arcgis-pro/.

[13] "Python Package Manager: how do I install 'mido' - a package that isn't listed?" Esri Community Forum, 8 Feb. 2024, https://community.esri.com/t5/python-questions/python-package-manger-how-do-i-instm-p/1379986#M69816.

[14] Russo, Matt. "Sonification with Python - How to Turn Data Into Music w Matt Russo (Part 1)." YouTube, 2 May 2022, https://www.youtube.com/watch?v=DUdLRy8i9qI.

[15] "Usage." Python-rtmidi Documentation, GitHub Pages, 2023, https://spotlightkid.github.io/python-rtmidi/usage.html.

[16] PySineWave. Python Package Index (PyPI), https://pypi.org/project/pysinewave/.

[17] Anderson, Jim. "An Intro to Threading in Python." Real Python, https://realpython.com/intro-to-python-threading/.

[18] Palmer, Moses."Pynput." 2023, pynput.readthedocs.io/en/latest/keyboard.html.

[19] Dahmm, Hayden, et al. 3D Printing to Create Models for the Blind, 30 Oct. 2012.

[20] "Special Awards." Swarthmore College, 2015, www.swarthmore.edu/commencement-2015/special-awards.

Appendices

Appendix A: Questions for Assessing User Understanding of Sonified Maps

Map 1: Crop Field Damage After a Hail Storm

1. On line 1, which field has the least damage?
2. On line 4, which two fields had the same amount of crop damage?
3. In line 8, which fourth of the line has fields with roughly the most crop damage?
4. In line sixteen, does the field with the most crop damage occur in the 1st fourth, 2nd fourth, 3rd fourth, or 4th fourth of the line?
5. Which line has the field with the greatest crop damage, line 40 or line 41?
6. On average, do the lines at the top or the bottom of the map have more fields?
7. Are the fields with the most damage located at the top or the bottom half of the map?
8. Do the lines near the top or the lines near the bottom have fields that tend to go from more damage to less damage as you move from left to right?
9. Between which corners of the map did the hail storm pass?

Map 2: Swarthmore's Geoexchange Wellfield

1. How many boreholes are on line 1?
2. On line 6, how many boreholes are in the first 12.5% of the line?
3. On line 14, which of two boreholes are the greatest distance apart?
4. Which has more boreholes, line 19 or line 20?
5. Are the lines with the most boreholes concentrated at the top, middle, or the bottom of the map?

Map 1 Answer Key

1. The fifth field
2. The third and the fourth field
3. The first fourth
4. The first fourth
5. 41
6. Top of the map
7. Top of the map
8. Top of the map
9. Between the NW and SE corners of the map

Map 2 Answer Key

1. One
2. Two
3. The fifth and sixth boreholes
4. Line 20
5. The middle

Appendix B: Test Product Code

```
#!/usr/bin/env python
# coding: utf-8

# TITLE: From Visual Data to Auditory Landscapes in ArcGIS
# AUTHOR: Grace Hegland

from pynput import keyboard
from pynput.keyboard import Key, Listener
from pysinewave import SineWave
from threading import Thread
from gtts import gTTS

import sys
import time
import winsound # this is a Windows specific library
import rtmidi
import pandas as pd
import os

# Read in the data that has been exported from an ArcGIS attribute table as a csv file
# This code can also be used outside of ArcGIS as long as the data is in a csv file
#filename = input("Enter the filename:")
#file_directory = input("Enter the file directory:")
#os.chdir(file_directory)
data = pd.read_csv(sys.argv[1])  # Load data as a pandas dataframe
numberOfRecords = len(data)
dataToMap = data.iloc[:,5] # data from column 5 of the pandas data frame

# General mapping function
# Calculate the proportional value of 'value' between 'min_value' and 'max_value'
# Map this proportional value to the range between 'min_result' and 'max_result'
def map_value(value, min_value, max_value, min_result, max_result):
    return min_result + (value - min_value)/(max_value - min_value)*(max_result - min_result)
```

```
# MIDI note numbers for the major pentatonic scale
note_midis = [36, 38, 40, 43, 45, 48, 50, 52, 55, 57, 60, 62, 64, 67, 69, 72, 74, 76, 79, 81]

if min(dataToMap) == max(dataToMap):
    midi_data = []
    for i in range(numberOfRecords):
        note = 57
        midi_data.append(note)
else:
    # Normalize data
    y_data = map_value(dataToMap, min(dataToMap), max(dataToMap), 0, 1)

    # Map y data to MIDI note numbers
    midi_data = []
    for i in range(numberOfRecords):
        note_index = round(map_value(y_data[i],0,1,0,19))
        midi_data.append(note_midis[note_index])

# Create a dictionary with longitude and latitude pairs as the keys and notes as the values
latLongNotes = dict(zip(zip(data['XLong'], data['YLat']), midi_data))

# Sort the dictionary first by decreasing latitude and then by increasing longitude
sortedLatLongNotes = dict(sorted(latLongNotes.items(), key=lambda item: (-item[0][1], item[0][0])))

# Create a list of the sorted latitudes
lats = [lat for _, lat in sortedLatLongNotes.keys()]

# Sort the dictionary by increasing longitude
sortedByLong = dict(sorted(latLongNotes.items(), key=lambda item: (item[0][0])))
longitudes = [long for long, _ in sortedByLong.keys()]

# Return true if the latitude is in the specified range of the current latitude (global)
# and if the longitude is in the range set by the zoom
# This range will define the width of our latitude bands when used to filter sortedLatLongNotes
def in_range(latlong_and_note, bandWidth):
    latlong, _ = latlong_and_note
    long , lat = latlong

    return lat <= currentLat and lat > currentLat - bandWidth and long >= longrange[0] and
long <= longrange[1]

# This function should read to the user what the current line is
def say_line(line_counter):
```

```
    myText = f"line {line_counter}"
    language = 'en'
    output = gTTS(text = myText, lang = language, slow = False)
    output.save("output.mp3")
    os.system("start output.mp3")

# This function should read to the user the function of each key
def say_help_info():
    Text = "Tab plays a line. The left arrow decreases the speed of lines. The right arrow increases the
    language = 'en'
    output = gTTS(text = Text, lang = language, slow = False)
    output.save("output.mp3")
    os.system("start output.mp3")

def calculate_zoom_percentage(zoom_counter):
    return 100/(2**(zoom_counter))

# This function should read to the user the current zoom level
def say_zoom_info(zoom_counter):
    ZText = f"The zoom level is {calculate_zoom_percentage(zoom_counter)} percent"
    language = 'en'
    output = gTTS(text = ZText, lang = language, slow = False)
    output.save("output.mp3")
    os.system("start output.mp3")

# Play a sine wave that decreases in pitch as you move along the x axis
def play_sine_wave(lineDuration):
    sinewave = SineWave(pitch=0, pitch_per_second= 5/lineDuration) # Starting pitch = 0
    sinewave.play()
    sinewave.set_pitch(-5) # Ending pitch
    time.sleep(lineDuration) # Total time of the sine wave in seconds
    sinewave.stop()

# Calculate the length of the delay between piano notes
def delayCalc(long1, long2, delayTime, intervalWidth):
    return ((long2 - long1)/intervalWidth)*delayTime

def notesToDelays(sortedLatLongNotes, delayTime, intervalWidth):
    delays = []
    longs = [long for long, _ in sortedLatLongNotes.keys()]
    for i in range(len(sortedLatLongNotes)-1):
        # Calculate the delays between consecutive notes using delayCalc
        # Append the calculated delays to the delays list
        delays.append(delayCalc(longs[i], longs[i+1], delayTime, intervalWidth))
    return delays
```

```
def midi_play(note, noteDuration):
    # 0x94 is the status byte for the "note on" message, note pitch, note velocity (volume) = 127
    note_on = [0x94, note, 127]
    # 0x84 is the status byte for the "note off" message, note pitch, note velocity (volume) = 0
    note_off = [0x84, note, 0]
    midi_out.send_message(note_on)
    time.sleep(noteDuration) # Play note for note duration before turning off
    midi_out.send_message(note_off)

# Play MIDI notes for their note durations with time delays between the notes
def play_midi_notes(sortedLatLongNotes, noteDuration, delayTime):
    timeDelays = notesToDelays(sortedLatLongNotes, delayTime, intervalWidth)
    for i in range(len(sortedLatLongNotes)):
        midi_play(list(sortedLatLongNotes.values())[i], noteDuration)
        if i < len(timeDelays):
            # If there are corresponding time delays, wait before playing the next note
            time.sleep(timeDelays[i])

def edgeDelays(longInitial, longFinal, delayTime, intervalWidth):
    # Calculate the length of the delay between the start of the line and the first note
    initialDelay = ((longInitial - longitudes[0]) / intervalWidth) * delayTime
    # Calculate the length of the delay between the last note and the end of the line
    finalDelay = ((longitudes[-1] - longFinal) / intervalWidth) * delayTime
    return (initialDelay, finalDelay)

def linePlay(currentLat, beep_frequency, lineDuration, noteTime):
    # Use the in_range function to filter the sorted note dictionary
    latitudeBands = dict(filter(lambda item: in_range(item, bandWidth), sortedLatLongNotes.items()))
    # Sort the latitude bands by increasing longitude
    sortedLatitudeBands = dict(sorted(latitudeBands.items(), key = lambda item : item[0]))
    noteCount = len(sortedLatitudeBands)
    longs = [long for long, _ in sortedLatitudeBands.keys()]
    delayTime = lineDuration - noteTime

    if noteCount > 0:
        # Use threading to simultaneously play the sine wave and the piano notes
        sine_thread = Thread(target = play_sine_wave, args = (lineDuration,))
        sine_thread.start()

        noteDuration = noteTime / noteCount
        initialDelay, finalDelay = edgeDelays(longs[0], longs[noteCount-1], delayTime, intervalWidth)
        time.sleep(initialDelay)
        play_midi_notes(sortedLatitudeBands, noteDuration, delayTime)
        time.sleep(finalDelay)
```

```
    else:
        # if there are no notes in the line, just play the sine wave
        play_sine_wave(lineDuration)

    winsound.Beep(beep_frequency, 500)

def speed_up():
    global lineDuration, noteTime
    lineDuration = lineDuration / 2
    noteTime = noteTime / 2

def slow_down():
    global lineDuration, noteTime
    lineDuration = lineDuration*2
    noteTime = noteTime*2

def on_key_release(key):
    global currentLat, beep_frequency, line_counter, longrange, zoom_counter, bandWidth,
    lineDuration, noteTime

    if currentLat < lats[-1]: #if current latitude is less than the minimum latitude
        return False # Stop the listener

    try:
        # Zoom into the first half of the region
        if key.char == "1":
            longrange[1] = longrange[0] + ((longrange[1] - longrange[0]) / 2) # Set a new upperbound
            zoom_counter += 1

        # Zoom into the second half of the region
        elif key.char == "2":
            longrange[0] = longrange[0] + ((longrange[1] - longrange[0]) / 2) # Set a new lowerbound
            zoom_counter += 1

        # Reset to the default zoom level
        elif key.char == "0":
            longrange=[longitudes[0], longitudes[-1]] # Original longitude range
            zoom_counter = 0

        # Read the line counter
        elif key.char == "i":
            say_line(line_counter)

        # Go to the starting line
        elif key.char == "s":
```

```
            longrange=[longitudes[0], longitudes[-1]]
            currentLat = lats[0]
            beep_frequency = 1000 # Initial beep frequency
            line_counter = 1
            zoom_counter = 0

        # Go to the middle line
        elif key.char == "m":
            longrange=[longitudes[0], longitudes[-1]]
            currentLat = lats[0] - bandWidth*29
            beep_frequency = 1000 - 5*29
            line_counter = 30
            zoom_counter = 0

        # Go to the ending line
        elif key. char == "e":
            longrange=[longitudes[0], longitudes[-1]]
            currentLat = lats[-1]
            beep_frequency = 1000 - 5*59
            line_counter = 60
            zoom_counter = 0

        # Read the zoom_counter
        elif key.char == "z":
            say_zoom_info(zoom_counter)

        # Read the help info
        elif key.char == "h":
            say_help_info()
    except:
        # Repeat the current line
        if key == Key.left:
            slow_down()

        elif key == Key.right:
            speed_up()

        # Move up a line
        elif key == Key.up:
            longrange=[longitudes[0], longitudes[-1]]
            currentLat += bandWidth
            beep_frequency += 5
            line_counter -= 1
            zoom_counter = 0
```

```
        # Move doum a line
        elif key == Key.down:
            longrange=[longitudes[0], longitudes[-1]]
            currentLat -= bandWidth
            beep_frequency -= 5
            line_counter += 1
            zoom_counter = 0

        # Play the line
        elif key == Key.tab:
            linePlay(currentLat, beep_frequency, lineDuration, noteTime)

        elif key == Key.esc:
            return False #Stop the listener

# Run these lines every time you want to run the code
midi_out = rtmidi.MidiOut() # Create a MIDI object to send MIDI messages to
midi_out.open_port(0) # Send MIDI message to output port

# Initialize all variables
currentLat = lats[0]
beep_frequency = 1000
line_counter = 1
longrange=[longitudes[0], longitudes[-1]]
zoom_counter = 0
intervalWidth = longitudes[-1] - longitudes [0]
bandWidth = (lats[0]-lats[-1])/60 # Fix the line number at 60 and calculate the bandwidth
lineDuration = 10
noteTime = 5

# Have the listener always listening with on_key_release as the handler
with midi_out:
    with Listener(on_release=on_key_release) as listener:
        listener.join()
```