Hearing Iterative and Recursive Behavior

Sonification Improves Student Understanding

Joel C. Adams
Dept. of Computer Science
Calvin University
Grand Rapids, MI, USA
adams@calvin.edu

Hayworth Anderson
Dept. of Computer Science
Calvin University
Grand Rapids, MI, USA
hayworth99@yahoo.com

ABSTRACT

Abstract topics such as recursion are challenging for many computer science students to understand. In this experience report, we explore *function sonification*—the addition of sound to a function to communicate information about the function's behavior in real-time as it runs—as a pedagogical approach for improving students' understanding of recursion. We present several example iterative and recursive function sonifications, plus spectrograms that illustrate their different sonic behaviors. We also present experimental evidence that using these sonifications significantly improved the understanding of recursion for students who used them, compared to students who used silent (i.e., traditional) versions of the same functions. Based on these experiences, we believe sonification has under-appreciated potential for teaching abstract computing topics.

CCS CONCEPTS

• Human-centered computing ~Auditory feedback • Social and professional topics ~Computer science education

KEYWORDS

Algorithm, audio, hearing, function, learning, media, recursion, sonic, sonification, sound

ACM Reference format:

Joel Adams and Hayworth Anderson. 2024. Hearing Iterative and Recursive Behavior: Sonification Improves Student Understanding. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024), March 20-23, 2024, Portland, OR, USA. ACM, New York, NY, USA, 7 pages.* https://doi.org/10.1145/3626252.3630866

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA

ACM 979-8-4007-0423-9/24/03
https://doi.org/10.1145/3626252.3630866

1 INTRODUCTION

We have noticed that many of our students wear earbuds or headphones while working in our computer labs, but most of the computer programs used in computer science (CS) education run silently. This led us to wonder: Might this be a missed opportunity? Might it be possible to leverage the use of sound to improve student understanding of computing abstractions that students find especially challenging?

As one example, many CS educators have noted that their students find the topic of recursion difficult to understand [7][9][11][21]. If students enjoy sonic stimulation, might it be possible to leverage sound in a way that helps them understand abstractions like recursion?

This line of thinking prompted us to explore **function sonification**—the addition of sound to a function to communicate information about the function's behavior in real-time as it runs. If the function contains a loop, sonification can let the user hear that loop executing. If the function is recursive, the user can hear its recursive behavior.

To create function sonifications, we used the Thread Safe Audio Library (TSAL) [1]. This let us easily add sound to C++ functions commonly used to teach iteration and recursion, turning calls to those functions into sonifications.

This paper is an experience report of our exploration. In Section 2, we present several examples of sonifications—both iterative and recursive. These examples illustrate the process of adding TSAL library calls to an existing function to make it generate appropriate tones in real time as the function executes. For each of these examples, we also present spectrograms that depict the function's sonic behavior. In Section 3, we describe an experiment we conducted to assess the effects of these sonifications on student learning. As we shall see, this experiment provides evidence that sonifications can significantly improve student understanding of recursion. In Section 4, we discuss previously published work that is related to our work. In Section 5, we present our conclusions and future plans.

2 SONIFICATIONS

As described in [1], TSAL makes it quite easy to turn a legacy program into a sonification. Within the program, one must perform four straightforward steps:

1. Define a **`Mixer`** object: a software representation of a multichannel mixer like those DJs use to mix sounds.
2. TSAL sounds are played by synthesizers, so define a **`Synth`** (or for multithreading, a **`ThreadSynth`**) object that can be used to generate sounds.
3. Add the **`Synth`** object to the **`Mixer`**. (For multithreaded sonifications—which we do *not* use here—a TSAL **`Mixer`** may mix the outputs of different **`ThreadSynth`** synthesizers, each with potentially different sonic characteristics.)
4. Use the **`Synth`** (or **`ThreadSynth`**) object to play sounds by invoking its **`play()`** method. For a sonification, the sounds played should reflect the program's algorithmic behavior.

Each sonification that follows assumes steps 1-3 were performed as follows (a sonification performs step 4 itself):

```
Mixer mixer = new Mixer();     // step 1
Synth synth = new Synth();     // step 2
mixer.add(synth);              // step 3
```

After these steps have been performed, a function **`f(int n)`** can be transformed into a sonification **`f(int n, Synth s`**). When that function is recursive, it can invoke **`s.play()`** to play a tone scaled to **`n`**—playing higher-pitched tones for higher values of **`n`**; lower-pitched tones for lower **`n`**-values.

2.1 Factorial

Figure 1 shows a C++ *iterative* factorial function sonification, with the added sonification code highlighted in blue:

```
long long factorial(unsigned n, Synth& synth) {
  long long answer = 1;
  for (unsigned i = 1; i <= n; ++i) {
    MidiNote note = scaleToNote(i, 0, 24, C3, C6);
    synth.play(note, Timing::MILLISECOND, 500);
    answer *= i;
  }

  return answer;
}
```

Figure 1: Iterative Factorial Sonification Function

TSAL employs standard MIDI notation, so in Figure 1, **`C3`** is a TSAL-defined constant for the C-note one octave below middle C (C4), and **`C6`** is a constant for the C-note two octaves above middle C. The TSAL utility function call:

```
scaleToNote(i, 0, 24, C3, C6)
```

returns a MIDI note from within the 3-octave range **`C3..C6`**, whose pitch is scaled to the position of **`i`** within the range **`0..24`**. (24 factorial will generate a numeric overflow, even using **`long long`** integer variables.) Once that MIDI note has been generated, the method call:

```
synth.play(note, Timing::MILLISECOND, 500);
```

plays that note for 500 milliseconds. The overall effect is that each iteration of the loop, the sonification plays a note whose sonic pitch (i.e., frequency) is scaled to the current value of the loop's control variable **`i`**.

To give the reader a sense of what one hears, we use a **spectrogram**—a sound-chart that graphs sonic pitch (Hz, vertical axis) against time (fractional seconds, horizontal axis). Figure 2 shows the spectrogram generated by the function in Figure 1 for the call **`factorial(10, synth)`**. From left-to-right, each 'step up' in Figure 2 represents the note played for variable **`i`** during an iteration of the for loop in Figure 1.

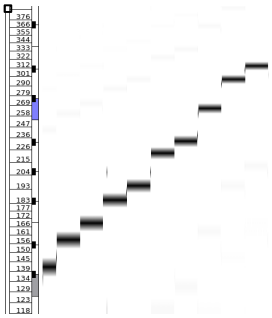


Figure 2. Spectrogram of Iterative factorial(10, synth)

In constrast with Figure 1, Figure 3 presents a sonification of a *recursive* factorial function:

```
long long factorial(unsigned n, Synth& synth) {
  long long answer = 0;
  MidiNote note = scaleToNote(n, 0, 24, C3, C6);
  synth.play(note, Timing::MILLISECOND, 500);

  if (n <= 1) {                    // base case
    synth.stop(Timing::MILLISECOND, 500);
    answer = 1;
  } else {                         // induction step
    answer = n * factorial(n-1, synth);
  }

  synth.play(note, Timing::MILLISECOND, 500);
  return answer;
}
```

Figure 3: Recursive Factorial Sonification

Note that the sonification in Figure 3 plays a note scaled to **`n`** *before* the recursive call—during the "winding" phase of the recursion—and then again *after* the recursive call—during the "unwinding" phase. To separate these two notes when the base case is reached, we use the **`Synth`** class **`stop()`** method to create a short break in the sound; otherwise, those two base case notes blend together into a single tone. Figure 4 is a spectrogram of the sounds that are generated when the sonification in Figure 3 computes **`factorial(10, synth)`**.

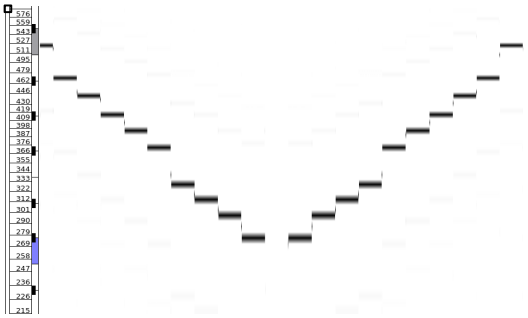


Figure 4. Spectrogram of Recursive factorial(10, synth)

Figures 2 and 4 allow the behavioral differences to be clearly *seen*, but when one runs the sonifications in Figures 1 and 3, those differences can be clearly *heard*—ascending tones vs. descending+ascending tones. In particular, the descending tones in Figure 4 are generated as the recursion "winds" toward the base case; the ascending tones are generated as the recursion "unwinds", allowing the user to hear the full time-complexity of the recursive version of the function.

Figures 2 and 4 provide visual representations of the distinctive sound patterns the functions in Figure 1 and 3 produce; we call such patterns the functions' *sonic signatures*.

To improve efficiency, we might revise the function in Figure 3 to use tail recursion. To conserve space, we leave this as an exercise for the reader; Figure 5 presents the spectrogram produced by a tail-recursive version of Figure 3:

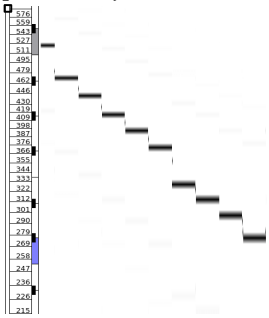


Figure 5. Tail-Recursive factorial(10, synth) Spectrogram

Sonification thus lets a user hear the behavioral differences of iterative and recursive solutions to the same problem, as well as the differences between tail and non-tail versions of a recursive function. Put differently, each version of **`factorial()`** has a distinct sonic signature.

2.2 Searching

Searching is an important problem for which different algorithms exist, including **linear search** and **binary search**. Figure 6 presents an iterative linear search function sonification that searches an array **`arr`** for a target value **`x`**:

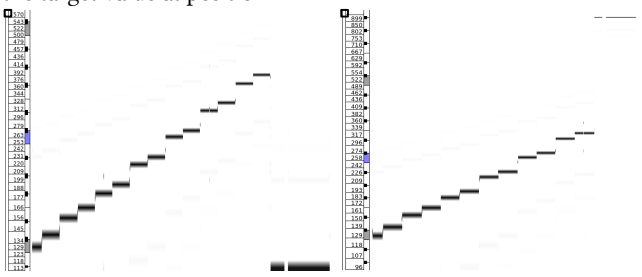
```
int linSearch(int arr[], int n, int x, Synth& synth) {
  for (int i = 0; i < n; ++i) {
    MidiNote note = scaleToNote(i, 0, MAX, C3, C6);
    synth.play(note, Timing::MILLISECOND, 500);
    if (arr[i] == x) {
      playSuccessNotes(synth);
      return i;
    }
  }
  playFailureNotes(synth);
  return -1;
}
```

Figure 6: Iterative Linear Search Sonification

Figure 6 invokes two special utility functions we created:

- **`playSuccessNotes()`**, that sounds two high-pitched 'fanfare' notes to indicate a search has succeeded, and
- **`playFailureNotes()`**, that sounds two low-pitched 'raspberry' notes to indicate that a search has failed.

Figure 7 presents spectrograms generated by Figure 6 when the search of an array of 14 elements (a) fails; and (b) succeeds, finding the target value at position 11:



(a) **(b)**

Figure 7. Spectrograms of Iterative Linear Searches

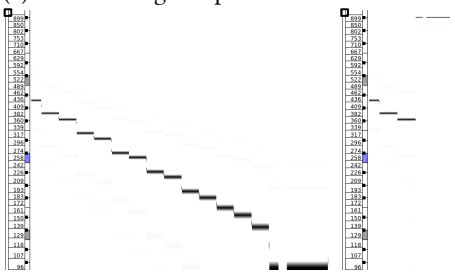
The thick lines at the bottom-right corner of Figure 7a represent the 'failure' notes; the thin lines at the top-right corner of Figure 7b represent the 'success' notes.

As an alternative to Figure 6, Figure 8 presents a sonfication of a tail-recursive linear search algorithm:

```
int linSearch(int arr[], int n, int x, Synth& synth) {
  if (n < 1) {                        // base case 1
    playFailureNotes(synth);
    return -1;
  }
  MidiNote note = scaleToNote(n, 0, MAX, C3, C6);
  synth.play(note, Timing::MILLISECOND, 500);
  int lastIndex = n - 1;
  if (arr[lastIndex] == x) {          // base case 2
    playSuccessNotes(synth);
    return lastIndex;
  } else {                            // induction step
    return linSearch(arr, lastIndex, x, synth);
  }
}
```

Figure 8: Recursive Linear Search Sonification

Figure 9 presents spectrograms generated by Figure 8 when searching an array of 14 elements for a target value: (a) fails, and (b) finds the target at position 11 in the same array:



(a) **(b)**

Figure 9. Spectrograms of Tail-Recursive Linear Searches

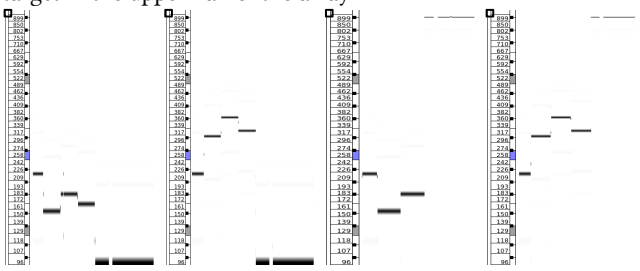
Figures 7 and 9 indicate how sonification lets one hear the front-to-back vs. back-to-front differences in the iterative and recursive approaches. One can also hear how the linear search algorithm's time to find the target varies, depending on the position of the target value within the array.

Linear search may be used on any array, but for sorted values, the faster binary search algorithm can be used. Figure 10 presents a tail-recursive binary search sonification:

```
int binSearch(int arr[], int lo, int hi, int x,
                                        Synth& synth) {
  if (lo > hi) {                // base case 1
    playFailureNotes(synth);
    return -1;
  }
  int mid = (hi + lo) / 2;
  MidiNote note = scaleToNote(mid, 0, MAX, C3, C6);
  synth.play(note, Timing::MILLISECOND, 500);
  if (arr[mid] == x) {          // base case 2
    playSuccessNotes(synth);
    return mid;
  } else if (arr[mid] > x) {    // induction step 1
    return binSearch(arr, lo, mid-1, x, synth);
  } else {                      // induction step 2
    return binSearch(arr, mid+1, hi, x, synth);
  }
}
```

Figure 10: Recursive Binary Search Sonification

Figure 11 presents spectrograms of the function in Figure 10 using an array of 14 values: (a) failing to find a target "below" the middle value, (b) failing to find a target "above" the middle value, (c) finding a target in the lower half of the array, and (d) finding a target in the upper half of the array:



(a) (b) (c) (d)

Figure 11. Spectrograms of Recursive Binary Searches

Tail-recursive and iterative binary search functions access the same array entries, and thus have identical sonic signatures.

2.3 Hanoi Towers

The Hanoi Towers Problem is to move N concentric disks from a source needle A to a destination needle B using an auxillary storage needle C, without placing a larger disk on top of a smaller disk. Figure 12 presents a spectrogram of the sounds generated by the call **`move(3, 'A', 'B', 'C')`** to the function in Figure 13:

Figure 12. Spectrogram of Hanoi Towers, N==3

By playing a note scaled to the number of disks **n** before and after each recursive call, Figure 13 lets a user hear the recursion "winding" and "unwinding" in real-time, as the function outputs its results.

```
void move(int n, char src, char dest, char aux,
                                        Synth& synth) {
  MidiNote note = scaleToNote(n, 0, 24, C3, C6);
  synth.play(note, Timing::MILLISECOND, 500);
  synth.stop(Timing::MILLISECOND, 5);
  if (n <= 1) {                     // base case
    cout << "Move the top disk from " << src
         << " to " << dest << endl;
  } else {                          // induction step
    move(n-1, src, aux, dest);
    synth.play(note, Timing::MILLISECOND, 500);
    synth.stop(Timing::MILLISECOND, 5);
    move(1, src, dest, aux);
    synth.play(note, Timing::MILLISECOND, 500);
    synth.stop(Timing::MILLISECOND, 5);
    move(n-1, aux, dest, src);
  }
  synth.play(note, Timing::MILLISECOND, 500);
  synth.stop(Timing::MILLISECOND, 5);
}
```

Figure 13: Hanoi Towers Sonification

Sonification thus provides a tool that can be used to trace and communicate a function's behavior to users. Can this improve students' understanding of recursion?

3 ASSESSMENT AND DISCUSSION

To assess the effectiveness of sonification as a tool for teaching recursion, we formed this research question:

> **RQ**: Does sonification of iterative and recursive functions improve students' understanding of recursion?

To answer this question, the authors designed the controlled experiment described below. Before conducting it, we submitted it to our university's Institutional Review Board, and received approval to proceed.

3.1 Experiment

The authors invited CS2 (*Introductory Data Structures*) students to participate in an experiment, in return for extra credit. 22 students, none of whom had visual impairments, volunteered to participate and were randomly assigned to two groups (*Control*, *Audio*) of size 11. Each group was invited to the same computer lab the same night, but at a different time.

After a brief welcome and introduction, each group's session consisted of four 12-minute segments covering the factorial, linear search, binary search, and the Hanoi Towers problems, respectively. In each segment, students were:

a. Shown a presentation about the problem and its solution. For the factorial and search problems, students were shown both iterative and recursive solutions; for Hanoi Towers, only a recursive solution was shown.
b. Directed to run a program implementing that algorithm. During this time, a pseudocode algorithm for the problem was projected on a screen at the front of the lab, and the students were invited to use it as a reference.
c. Directed to re-run the program using different inputs.

In step b, the algorithms shown to the *Audio* group included TSAL sonification calls; the *Control* group algorithms did not.

Also in step b, each group was shown how to run a given program from the command-line. For example, to compute 5 factorial using the iterative version, the *Control* group entered:

```
$ ./factorialIter 5
```

The *Audio* group was instructed to run the same programs, but adding the **-a** (audio) switch and wearing earbuds:

```
$ ./factorialIter 5 -a
```

The *Control* group thus saw a program's normal output; the *Audio* group both heard the sonification and saw the output.

To answer our research question **RQ** based on students' long-term memories, we emailed each student a link to a 9-question, 10-point online quiz one week after their session. 8 of 11 *Control* group members and 11 of 11 *Audio* group members completed the quiz within 48 hours (required to receive the extra credit).

The nine quiz questions covered the following topics:

Q0 (not scored). Students were asked to rate how engaging they found their session on a 1 (*Boring*) to 10 (*Loved it*!) scale.

Q1 (1 pt). Correctly identify the definition of the word 'recursion' from among several multiple-choice options.

Q2 (1 pt). Given a definition for "base case", identify that phrase from among several similar options (e.g., "worst case").

Q3 (2 pts). Given the code of a tail-recursive function **`f1()`** that reverses an array's values, calculate the value the function returns when called with particular arguments.

Q4 (2 pts). Given the code of a non-tail recursive function **`f2()`** that also reverses an array's values, calculate the value it returns when called using the same arguments as in Q3.

Q5 (1 pt). Given a recursive function **`f3()`**'s code that sums an array's values and a call to it, calculate the value it returns.

Q6-8 (1 pt each). Given a recursive function **`f4()`** that performs linear search, calculate its return values for three different calls:
- Q6: the target value was not present in the argument-array;
- Q7: the target value was the last value in the array;
- Q8: the target value was the first value in the array.

3.2 Results and Discussion

Table 1 summarizes the groups' performances on Q1-Q8:

Question	Possible Points	Groups' Mean Scores	
		Control	Audio
Q1	1	1	1
Q2	1	0.125	0.455
Q3	2	1.25	1.5
Q4	2	0.25	1
Q5	1	0.375	0.545
Q6	1	0.5	0.909
Q7	1	0.625	1
Q8	1	0.5	0.818

Table 1: Quiz Analysis: Individual Questions and Overall

The *Audio* group thus did as well or better than the *Control* group on every question, especially Q4, Q6, Q7, and Q8.

Table 2 compares the groups' overall quiz performances:

Group	Max.	Min.	Median	Mean
Control	10	1	4	4.625
Audio	10	4	7	7.227

Table 2: Summary Statistics of Quiz Scores

The groups' quiz scores comprised a (roughly) normal distribution, so we compared their means using a two-tailed, two-sample equal-variance t-test, with 0.05 as our significance threshold. Our null hypothesis was that there would be no significant difference between the groups' mean scores, but the result of our t-test ($p=0.04391$) led us to reject the null hypothesis. **Sonification apparently significantly improved our *Audio* group students' understanding of recursion**.

On our unscored "engagement" question Q0, the groups' mean responses were 5.626 (*Control*) and 5.725 (*Audio*). This difference was not statistically significant ($p=0.90659$), so we *cannot* attribute the *Audio* group's better quiz performance to better engagement. But the differing quiz completion levels (8/11 vs. 11/11) might reflect different engagement levels.

3.3 Potential Validity Threat

We used random selection to assign students to our *Audio* and *Control* groups. Because of our modest group sizes, it is possible that stronger students were assigned to the *Audio* group and weaker students to the *Control* group by chance. Student privacy concerns prevented us from using scores, grades or similar information to equally distribute strong and weak students between the two groups, or assessing their classroom performance after the experiment. We also had no means of controlling the number of students who volunteered, aside from the participation-incentive we provided.

4 BACKGROUND MATERIAL

This section explores previous work related to this paper.

4.1 The Difficulty of Recursion

The literature contains many papers that explore *why* students find recursion difficult; a small sample includes:

- Götschi, et al [9] studied students' mental models of recursion, identifying some as valid, others as invalid.
- Sanders, et al [19][20] also examined such mental models, noting that some models let students correctly evaluate a recursive function without understanding recursion.
- Mirolo [17] explored computing competency dimensions, noting that abstraction contributes to recursion difficulty.
- Sooriamurthi [21] named three issues that keep students from grasping recursion: (i) lack of exposure to declarative thinking, (ii) not understanding the functional abstraction, and (iii) inability to express a recursive solution.

Based on our experimental results, we believe that students who *hear* recursive behavior in real time receive extra information via their auditory channel that helps them build more accurate mental models of recursive behavior than students who hear nothing when a recursive function executes.

4.2 Pedagogical Proposals For Recursion

There have also been many papers describing pedagogical proposals for teaching recursion, including these:

- Augenstein and Tenenbaum [2] first proposed the use of non-recursive and recursive solutions to the same problem, which we utilized in Figures 1 and 3, and 2 and 8.
- Ginat and Shifroni [8] argued that a declarative, abstract, functional approach enhances students' recursive thinking.
- Kruse [14] proposed the use of activation-tree diagrams for tracing the behavior of recursive calls to help students decide when to use or not use recursion.
- Liss and McMillan [15] argued for using maze-navigation with backtracking as an example for teaching recursion.

However, a recent survey of the literature on teaching recursion notes that "there is a surprising lack of evidence for the effectiveness of many of the methods presented" [16].

The pedagogical benefits of visualization are well-known; Hundhausen, et al [12] and Naps, et al [18] provide excellent overviews. There is also an extensive body of work devoted to using visualizations to help students understand recursion. A sampling of this work includes:

- Dann, et al [5] described using 3D recursive animations to help students see and understand recursive behavior.
- Stern and Naish [22] explored visualizations of recursive data structures algorithms, providing a classification scheme based on how the data structure is traversed.
- Stephenson [23] presented three pedagogical examples (fractals, flood-fill, maze) that his students found engaging.
- Velázquez-Iturbide, et al [24] described a Java tool that animates recursive traces, activation trees, and call stacks.

Our work differs from such papers in focusing on the use of a different sense (hearing) for teaching recursion, and by providing evidence that it improves student learning.

Note that our results do not allow us to make any claims about *how much* sonification improves learning compared to visualization or other pedagogical approaches; such comparisons will require additional experimentation.

4.3 Sonification and TSAL

Kramer, et all [13] define **sonification** as "the transformation of data relations into perceived relations of an acoustic signal for the purposes of facilitating communication or interpretation." Sonification thus differs from *aurelization*—the generation of sounds by software in general—as described by DiGiano and Baecker [6], and from *earcons* (i.e., auditory icons) of Blattner, et al [3]. Our work differs by focusing on pedagogy, and by using a current, object-oriented language.

Our work builds on previous work by Adams, et al [1]. That paper described TSAL, presented several sorting sonifications, and demonstrated their benefits for improving student learning. We generalize on that work by exploring the broader topic of *function sonification*, specifically the pedagogical impact of *iterative and recursive function sonification.*

5 CONCLUSIONS AND FUTURE WORK

This paper presents an experience report on the use of *function sonification* as a pedagogical tool. We have presented a group of function sonifications—iterative and recursive—plus spectrograms that illustrate these functions' distinct *sonic signatures.* These signatures allow a user to hear different functions' behavioral differences: iterative vs recursive, linear search vs. binary search, and so on.

We have also presented controlled experimental results that indicate sonifications improved students' understanding of recursion. We find this improvement especially interesting because TSAL was designed to aid visually impaired students [1] and none of our students had impaired vision.

Our *Audio* and *Control* groups did *not* report significantly different levels of engagement. Given the differences in the groups' quiz performances, this appears to contradict Naps, et al [18] who argue that a technology's pedagogical value depends on how engaging it is. But different engagement levels could explain why just 8 of 11 *Control* students completed the quiz, compared to 11 of 11 *Audio* students.

Sonification may also be useful for teaching other topics. For example, we have created sonifications of the in-order, pre-order, and post-order binary search tree traversals but have not yet evaluated their pedagogical impact; these may be the subject of a future paper. Other possibilities include:

- Auditory alerts: generating distinct sonic feedback when exceptions are thrown. Such feedback is useful even when a user is not looking at their computer's screen.
- Hearing the differences between single-threaded and multi-threaded versions of an algorithm.

Sonification thus opens up many possibilities for future exploration, with the potential to reveal new insights and help students better understand computing abstractions.

The sonifications we have created map data values to sonic frequencies, which seemed like an intuitive way to represent recursive and iterative function behaviors. However, other sonification strategies are possible, including:

- Varying the *loudness* (i.e. amplitude) of sounds in response to different data values, and/or
- Varying the *time* (i.e., duration) of sounds and/or silences to communicate information about data values.

We hope to explore these approaches in the future, as our initial experiences with sonification lead us to believe that it has significant under-utilized potential for helping students understand abstract computing concepts.

For anyone interested in repeating our experiments, our sonifications, experimental materials, and data are freely available by request to the authors.

ACKNOWLEDGMENTS

This work was made possible by U.S. NSF-DUE grant #1822486, and by TSAL, which may be freely downloaded from Github, as indicated in [1]. The spectrograms in Section 2 were created using Sonic Visualizer [4].

REFERENCES

[1] J. Adams, B. Allen, B. Fowler, M. Wissink. The Sounds of Sorting Algorithms: Sonification as a Pedagogical Tool, *Proc. of the 53rd ACM SIGCSE Technical Symposium on Computer Science Education* (SIGCSE'22), Feb 2022. pp. 189-195.

[2] M. Augenstein and A. Tenenbaum. A Lesson in Recursion and Structured Programming. *Proc. of the ACM SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '76), Feb 1976, pp. 17–23.

[3] M. Blattner, D. Sumikawa and R. Greenberg, Earcons and Icons: Their Structure and Common Design Principles. SIGCHI Bulletin 21, 1, July 1989, pp. 123–124.

[4] C. Cannam, C. Landone, and M. Sandler. Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files, *Proceedings of the ACM Multimedia 2010 International Conference*, Oct 2010, pp. 1467-1468.

[5] W. Dann, S. Cooper, and R. Pausch. Using Visualization to Teach Novices Recursion. *Proc. of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE'01). June 2001, pp. 109–112.

[6] C. DiGiano and R. Baecker. Program auralization: Sound enhancements to the programming environment. *Proc. of the Conference on Graphics Interface '92.* Morgan Kaufmann Publishers, Sept 1992, pp. 44-52.

[7] M. Endres, W. Weimer, and A. Kamil. An Analysis of Iterative and Recursive Problem Performance. *Proc. of the 52nd ACM Technical Symposium on Computer Science Education* (SIGCSE '21), Mar 2021, pp. 321–327.

[8] D. Ginat and E. Shifroni. Teaching Recursion in a Procedural Environment—How Much Should We Emphasize the Computing Model? *Proc. of the 30th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '99). Mar 1999, pp. 127–131.

[9] D. Ginat. Do Senior CS Students Capitalize on Recursion? *Proc. of the 9th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE'04), June 2004, pp. 82–86.

[10] T. Götschi, I. Sanders, and V. Galpin. Mental Models of Recursion. *Proc. of the 34th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE'03), Mar 2003, pp. 346–350.

[11] B. Haberman and H. Averbuch. The case of base cases: why are they so difficult to recognize? student difficulties with recursion. *Proc. of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE'02), June 2002, pp. 84–88.

[12] C. Hundhausen, S. Douglas, and J. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing* 13(3), June 2002, pp. 259–290.

[13] G. Kramer, B. Walker. T. Bonebright, P. Cook, J. Flowers, N. Miner and J. Neuhoff. "Sonification Report: Status of the Field and Research Agenda" (2010). *Faculty Publications, Department of Psychology, University of Nebraska-Lincoln.* Online, accessed 2021-12-01: https://digitalcommons.unl.edu/psychfacpub/444.

[14] R. Kruse. On Teaching Recursion. Proc. of the 13th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'82), Feb 1982, pp. 92–96.

[15] I. Liss and T. McMillan. An Amazing Exercise in Recursion for CS1 and CS2. *Proc. of the 19th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '88). Mar 1988, pp. 270–274.

[16] S. Mackay. 2022. What Does Literature Tell Us About Recursion? *Proc. of the 53rd ACM Technical Symposium on Computer Science Education* (SIGCSE'22), Mar 2022, p. 1173.

[17] C. Mirolo. Learning (Through) Recursion: a Multidimensional Analysis of the Competences Achieved by CS1 Students. *Proc. of the 15th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE '10), June 2010, pp. 160–164.

[18] T. Naps, G. Roessling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. SIGCSE Bulletin 35 (2), June 2003, pp. 131-152.

[19] I. Sanders, V. Galpin, and T. Götschi. Mental models of Recursion Revisited. *Proc. of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (ITICSE'06), June 2006, pp. 138–142.

[20] T. Scholtz and I. Sanders. Mental Models of Recursion: Investigating Students' Understanding of Recursion. *Proc. of the 15th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE '10), June 2010, pp. 103–107.

[21] R. Sooriamurthi. Problems in comprehending recursion and suggested solutions. *Proc. of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE'01), June 2001, pp. 25-28.

[22] L. Stern and L. Naish. Visual Representations for Recursive Algorithms. *Proc. of the 33rd SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '02), Mar 2002, 196–200.

[23] B. Stephenson. Visual Examples of Recursion. *Proc. of the 14th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE'09, June 2009, p. 400.

[24] J. Velázquez-Iturbide, A. Pérez-Carrasco, J. Urquiza-Fuentes. 2008. SRec: An animation system of recursion for algorithm courses. *ACM SIGCSE Bulletin* 40(3), Sept 2008, pp. 225-229.