2021. *Proc Ling Soc Amer* 6(2). 5135. https://doi.org/10.3765/plsa.v6i2.5135

From "Hello, World!" to Fourier transformations: Teaching linguistics undergraduates to code in ten weeks or less

Reed Blaylock*

Abstract. I used Backward Design to scaffold ten weeks of assignments that taught students how to perform sine wave vowel synthesis and a Fourier transformation approximation using just a few fundamental programming concepts. This strategy gave all students, regardless of their previous programming experience, the opportunity to implement algorithms related to core concepts in phonetics and speech technology. Reflecting on the course, it seems that the coding assignments were generally well-received by students and contributed to students programming something complex and meaningful.

Keywords. phonetics; speech technology; backward design; scaffolding; coding

1. Introduction. As a graduate student, I served as teaching assistant for a general education undergraduate Speech Technology course taught in a Linguistics department for six terms and three different instructors of record. Students in this course learned the linguistic underpinnings, social impact, and commonly-used algorithms (conceptually) of speech synthesis and recognition technology, as well as how to program parts of speech technology algorithms. In my time as a teaching assistant, however, I noticed—though I am certainly not the first (e.g., Jenkins & Davy, 2002)—that students without previous programming experience were at a real disadvantage in the course. Even though we started teaching code from the fundamentals, students had relatively few chances to practice coding; and we never spent enough time on coding for students to be able to program an actual piece of speech technology (i.e., vowel synthesis or recognition).

In Spring 2020 I became instructor of record for a ten week junior-level course on Speech Technology. The course had originally been scheduled as face-to-face, but ended up being a completely asynchronous style of emergency remote teaching due to the COVID-19 pandemic. The students in the course were 17 Linguistics majors, all of whom had taken at least one introductory phonetics class as a prerequisite. They entered the class with a variety of programming experience: some had none, some had plenty, and some were concurrently enrolled in a Computational Linguistics class.

I used this opportunity to try to create a progression of coding assignments that would be simple enough to be accessible to students with no programming experience, meaningful enough that everyone felt they had learned new and valuable skills in the domain of phonetics and speech technology, and practical enough that students could try to get jobs in the speech technology industry (if they wanted).

2. Strategy. I used Backward Design (Wiggins & McTighe, 1998; Cho & Allan, 2005) to scaffold weekly programming assignments that culminated in a final summative project of implementing a speech synthesis algorithm and a speech recognition algorithm. For this course, speech synthesis was vowel synthesis—having a computer create a list of numbers that, when

* Thanks to the LSA's Faculty Learning Community on Scholarly Teaching for their support. Thanks also to Louis Goldstein, Khalil Iskarous, and Mary Byram-Washburn for creating and teaching the excellent Speech Technology courses I have had the pleasure to teaching-assist. Author: Reed Blaylock, University of Southern California (reed.blaylock@gmail.com).

played as audio by a computer, sounded to the human ear like a steady-state vowel. Speech recognition was a Fourier transform approximation—having a computer recognize a vowel by identifying which simple frequencies in the complex vowel signal are most prominent.

2.1. SCAFFOLDING AND BACKWARD DESIGN. Scaffolding (e.g., Wood et al. 1976; Verenikina 2008) requires a distal "end goal" learning objective that the teacher and learner are aiming for as well as more proximal "next step" learning objectives that are challenging but accomplishable for the learner. The teacher initially assists the learner as they become more familiar with each new skill; as the learner's competence with a skill increases, the teacher shifts their assistance toward the next skill on the path toward expertise. A scaffolding strategy therefore pairs well with the Backward Design technique of identifying learning objectives and developing appropriate lessons and assessments for those objectives. Backward Design begins with determining the "end goal" learning objectives of a course, then deconstructing those objectives to identify what skills and knowledge students would need to accomplish those objectives. These intermediate learning objectives can be further deconstructed recursively down to the skills that students are expected to already know when they start the course. Scaffolded teaching guides the student forward along the path of learning objectives outlined with Backward Design.

Scaffolding is typically described as a collaboration between learner and teacher. Teachers have to pay attention to each learner's skill level to determine when it is appropriate to remove scaffolding on one skill and add scaffolding to another. That way, learners are faced with challenges they are ready for without feeling that the pace of learning is too fast or too slow. However, given the asynchronous format of the course and difficulties with course preparation in the COVID-19 pandemic (see Blaylock et al. 2021), I felt unable to offer an appropriately flexible scaffolding experience to my students.

Instead, I aimed to introduce scaffolded coding skills at a pace that I hoped would be appropriately challenging for novice programmers. I did this with a set of weekly programming assignments to lead students from the very basics of programming—by tradition, getting a computer to output the statement "Hello, World!"—to implementing algorithms for synthesizing and identifying vowels. Each assignment had a "Read, write, reflect" structure: students read and explained samples of code, wrote algorithms using code they had read, and reflected on their work (Selby 2011; Zavala 2016). Each assignment's algorithm was useful in the scope of phonetics (e.g., synthesize one sine wave) and a component of the final assessment (e.g., synthesize a vowel).

2.2. Vowel SYNTHESIS AND FOURIER TRANSFORMATION APPROXIMATION. The course featured two major sections: speech synthesis and speech recognition. The summative assessment featured a synthesis task and a recognition task that could both be performed using just a few relatively simple coding concepts (i.e., lists and matrices, for loops, maybe a built-in function or two). Detailed descriptions of each assignment are given in the Appendix; the assignments themselves are available as supplemental material.

In this class, vowel synthesis was performed by adding simple sine waves (representing harmonics) together into a complex wave (the vowel). Simple sine waves were synthesized from scratch using mass-spring dynamical systems (Cromer 1981). Dynamical systems can be challenging for learners unused to the math, so I introduced them early in class and in code because they would show up thematically throughout the class—starting with sine waves, then later algorithms like dynamic programming (for concatenative synthesis) and stochastic gradient descent (for neural network speech recognition). In code, the core skills for this vowel synthesis included constructing for loops, creating lists, retrieving an element from a list, appending an

element to a list, and simple arithmetic—all of which are crucial skills for any novice programmer.

The speech recognition built on the coding techniques learned for vowel synthesis. I chose an approximation of a Fourier transformation that identifies which simple frequencies are most present in a complex wave—or in other words, an algorithm that identifies the harmonics of a vowel. This technique required three pieces: a vowel, a set of simple sine waves to compare against the vowel, and a comparison computation like the inner product. Students by this point had already learned how to create sine waves, and vowels, so the only major new coding tasks were 2-dimensional matrices for storing the sine waves—which are essentially just the lists we used in vowel synthesis—and a summing function required for the inner product.

A major coding skill emphasized throughout the whole course was commenting code. Modern computer programs are often created with two types of information: commands in a programming language meant for a computer to interpret and execute, and human-oriented text called "comments" that explain what the code is supposed to do. Consistent and informative comments are considered good coding practice (e.g., Drevik 1996). I demonstrated commenting technique with line-by-line instructions in the "Write" part of each assignment so students could focus on coding one step at a time instead of trying to assemble programs from scratch (a skill I was not teaching them). In the "Read" parts of assignments, students were usually asked to write their own comments to practice explaining code.

Sine waves and Fourier transformations are foundational components of phonetics and speech technology, and the coding techniques used for these algorithms (i.e., nested for loops and iteratively manipulating lists and matrices) are quite common in programming at all levels. The aim of the course was that by the time of the summative assessment, students would feel as though they had learned to code something useful.

2.3. SUMMATIVE ASSESSMENT. As I had planned through Backward Design, the summative assessment was an extension of the previous coding assignments. The task was to do some simple speech recognition by approximating a Fourier transformation—specifically, to compare the complex wave of a vowel against many simple signals and using the inner product to identify which simple signals were most prominent in the vowel. In addition to coding the Fourier transform approximation, students were also responsible for synthesizing the vowels that would be recognized and the simple signals that each vowel would be compared against.

Every part of the summative assessment had been introduced by a previous coding assignment (see the Appendix), but also required combining some coding skills in new ways. For example, students had to synthesize not one but *three* different vowels using the vowel synthesis technique from Assignment 6 and storing the vowel vectors in a 2-dimensional matrix as they had learned in Assignment 7. Students had not coded a Fourier transform approximation by this time (though they had learned about the strategy in short lecture videos), but the steps to do it were taken from Assignments 5, 7, and 9. In this way, students were assessed on their ability to apply coding techniques they had learned to new but familiar tasks.

In short, over the course of nine weeks, students encountered all the skills they needed to implement a reasonably complicated speech synthesis and recognition program. (This was unlike the previous courses I had been a teaching assistant for in which students were never expected to be able to program this way.)

3. Result. Students generally responded positively to the coding assignments in a graded, open-ended reflection activity that was assigned at the end of the term. I did not ask about the coding assignments specifically in this assignment, but instead invited students to write about whatever

impacted them the most in the course. Reflections from the 13 students who submitted them indicated that students were grateful for the coding experience and the pleasure of implementing course content as code. Between these generally positive reflections and overall good performance on the code assignments, I believe the learning was successful. Given the complexity of the code for vowel synthesis and Fourier transform approximation, I believe students learned an astonishing amount in this short time.

Specific feedback included the following:

- Several students appreciated having a new programming skill.
- Three students appreciated thinking about code differently; one of them was an experienced programmer who appreciated the emphasis on extensively commenting code, and wish they'd learned that skill earlier.
- A few students enjoyed learning the path-finding algorithms (coded in Assignment 8).
- Some experienced programmers found the early assignments too trivial.
- Two students appreciated the assignments with "tangible" results, like synthesizing an [i] vowel and coding a greedy search algorithm.
- One experienced programmer enjoyed making vowels from scratch, and looked forward to applying path-finding algorithms to games they make.
- One student appreciated the hands-on connection to topics they had learned about in an introductory phonetics class.
- One student enjoyed learning the equations and then implementing them in code.

Without prompting, two students mentioned the scaffolding of the assignments in their reflections: a student with little to no coding background reported surprise at how successfully they had assembled code from previous assessments they had found challenging; and, a student concurrently taking a computational linguistics course was grateful for how the extensive scaffolding in each assignment made it easier to understand the algorithm they were asked to implement.

Telling the students what to write line by line may sound to some like too much "hand-holding", but students still had to work hard and ask questions—especially for the more complex algorithms. Successful code was built on an understanding of MATLAB syntax and algorithms we had discussed in the abstract; students struggled when they misunderstood part of the algorithm being coded, or when they didn't connect previous code assignments to the one they were working on.

4. Instructor's reflection. Although a main motivation for this scaffolded approach was to make coding more accessible for students without programming experience or in precarious learning situations (pandemic-induced or otherwise), it seemed that these students were still more likely to struggle with code assignments than those who had coding experience and supportive learning environments. At the same time, some students who had coded before found the initial assignments annoyingly trivial. A future version of this class could benefit from letting students self-select into different "streams" that suit their level of experience, interest, and work bandwidth as suggested by Jenkins & Davy (2002): students would all learn together the same linguistic underpinnings, social impact, and commonly-used algorithms (conceptually) of speech synthesis and recognition technology; but what they would learn to code and how quickly they would be expected to learn it would depend from stream to stream.

For example, the less intensive stream for this course might lead students to sine wave vowel synthesis over the course of all ten weeks instead of being compressed into six; the more

advanced stream could move through code faster and encourage students to implement more complicated algorithms like dynamic programming and their own neural networks. Students in both streams would be expected to learn how the complicated algorithms work conceptually, but only students with previous programming experience would be expected to implement them. With multiple streams, students would face coding challenges more suitable to their current abilities (instead of a one-size-fits-nobody approach); this would help to ensure that students learn to code something meaningful that is also appropriate for their level.

In sum, thoughtful course design helped my students learn a practical set of coding skills in an impressively short time. But through this experience I was reminded that *thoughtfully flexible* course design would serve students even better (e.g., Rose & Meyer 2002; see also the discussion in Blaylock et al. 2021).

References

Blaylock, Reed, Evan D. Bradley, Ann Bunger & Taylor Sharp. 2021. Emergency remote teaching in linguistics during the early COVID-19 pandemic. *Proceedings of the Linguistic Society of America (PLSA)* 6(2). 5111. https://doi.org/10.3765/plsa.v6i2.5111.

Cho, Jeasik & Allen Trent. 2005. "Backward" curriculum design and assessment: What goes around comes around, or haven't we seen this before?. *Taboo* 9(2). 105.

Cromer, Alan. 1981. Stable solutions using the Euler approximation. *American Journal of Physics* 49(5) .455. https://doi.org/10.1119/1.12478.

Drevik, Steve. 1996. How to comment code. *Embedded Systems Programming* 9. 58–65.

Jenkins, Tony & John Davy. 2002. Diversity and motivation in introductory programming. *Innovation in Teaching and Learning in Information and Computer Sciences* 1(1).1–9. https://doi.org/10.11120/ital.2002.01010003.

Rose, David H. & Anne Meyer. 2002. *Teaching every student in the digital age: Universal design for learning*. Alexandria, VA: Association for Supervision and Curriculum Development.

Selby, Cynthia C. 2011. Four approaches to teaching programming. Learning, Media and Technology: A Doctoral Research Conference. London.

Verenikina, Irina M. 2008. Scaffolding and learning: Its role in nurturing new learners. University of Wollengong Faculty of Education – Papers (Archive). https://ro.uow.edu.au/edupapers/43.

Wiggins, Grant & Jay McTighe. 1998. What is backward design. *Understanding by Design* 1. 7–19. https://goglobal.fiu.edu/_assets/docs/whatisbackwarddesign-wigginsmctighe.pdf.

Wood, David, Jerome S. Bruner & Gail Ross. 1976. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry* 17. 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x.

Zavala, Laura. 2016. Read, manipulate, and write: A study of the role of these cumulative skills in learning computer programming. *Proceedings of the ASEE NE 2016 Conference*. 28–30.

Appendix I - Course Schedule

Below I present nine weeks of assignments as four "units", each of which emphasized a different set of skills. Unit 1 (Assignments 1-3) introduced the basics of programming in MATLAB, including lists and for loops. Unit 2 (Assignments 4-6) used the basic concepts from Unit 1 for sine wave and vowel synthesis using mass-spring dynamical systems. Unit 3 (Assignments 7-8) introduced multidimensional matrices as an extension of the lists from Unit 1. Finally, Unit 4

(Assignment 9) used the loops from Unit 1 and the matrices from Unit 3 to compare vectors from different matrices, a technique used for spectrogram comparison and a Fourier transform approximation.

In Unit 1 (Assignments 1 through 4), students were introduced to the coding environment (MATLAB Online) and common code techniques including the use of comments, lists, and for loops. These were foundational skills that would be used in every assignment afterward. Students were also introduced in these assignments to the practice of reading a chunk of code and analyzing it by attempting to predict its output, skills which are important both for understanding new code and debugging their own.

Assignment 1 tasks included:

- Open the coding environment (MATLAB Online)
- Open, edit, and save MATLAB file
- Upload a MATLAB file to the course learning management system

Assignment 2 tasks included:

- Read single lines of code that involve basic list operations (creating lists, appending list elements, retrieving a list element with numerical and variable indices, replacing a list element) and arithmetic operators.
- Write comments describing what each line of code does
- Predict the output of a code chunk

Assignment 3 tasks included:

- Read for loops that perform basic list operations and arithmetic
- Predict the output of for loops
- Write comments describing what each line of a for loop does
- Write code involving basic list operations and arithmetic based on comment instructions

Assignments 4 through 6 (Unit 2) culminated in students synthesizing their own [i] vowel. Their previous work with lists and for loops evolved into modeling spring and mass dynamical systems separately, including modeling formant transitions with the spring systems; then we synthesize a single sine wave with a mass-spring system, and finally the synthesize a whole vowel with the superposition of several mass-spring systems.

As planned through Backward Design, Assignments 1-5 were tailored to give students everything they needed to do vowel synthesis in Assignment 6. Their ability to do vowel synthesis in this way was later re-assessed (with minor changes) in the summative assessment.

Assignment 4 tasks:

- Read code implementations of dynamical systems and identify whether they are mass or spring systems, what the goal of the system is (if it's a spring system), and describe the updating function in words
- Write a sequence of dynamical spring systems (using for loops) that roughly approximate formant transitions in the phrase "I owe you"

Assignment 5 tasks:

- Read nested for loops, describe the the code in line-by-line comments, and predict the output of the loop
- Create simple sine waves by coding mass-spring dynamical systems

Assignment 6 tasks:

- Record your voice (an [i] vowel) in Praat
- Take a spectral slice of your [i] vowel

- Manually measure the frequency and amplitude of every harmonic in your spectral slice between 0-5000 Hz
- Synthesize your own [i] vowel by creating a simple sine wave (from a mass-spring system) for each harmonic and adding the sine waves together

Assignments 7-8 (Unit 3) introduce students to multidimensional data in the form of 2-dimensional, 3-dimensional, and 4-dimensional matrices, as well as the min() and sum() functions. The 4-dimensional matrices represent the costs of paths in a network used for concatenative synthesis, which students traverse by coding a greedy search algorithm (in which path choices are made by the min() function and the final path cost is calculated with the sum() function). Path-finding algorithms like the greedy search algorithm are crucial for understanding how concatenative synthesis works.

Assignment 7 tasks:

- Read, write, and predict the outputs of 2-dimensional, 3-dimensional, and 4-dimensional matrices
- Create a 4-dimensional matrix to represent a network that could be used for the best-path problem

Assignment 8 tasks:

- Read and predict the output of code that uses the min() and sum() functions for lists and 2-dimensional, 3-dimensional, and 4-dimensional matrices
- Code the greedy search algorithm for the best-path problem using a for loop, the min() and sum() functions, and a 4-dimensional matrix that represents path segment costs

Assignment 9 (Unit 4) was the last homework assignment before the summative assessment. It introduced the inner product computation (which involved the sum() function learned earlier) for calculating similarity between vectors; the inner product was then used to create a tiny speech recognition system that identified which of two spectrograms (represented as different 2-dimensional matrices) was more similar to a third by comparing the spectrograms moment-by-moment. The strategy of using the inner product to compare vectors of matrices was then a crucial component of the Fourier transform approximation in the summative assessment.

Assignment 9 tasks:

- Calculate the inner product between two lists using the max() and sum() functions
- Represent a spectrogram as a 2-dimensional matrix
- Evaluate the similarity of two spectrograms using the inner product in a for loop

I did not assign a code assignment in the tenth (and last) week of class in the hopes that students would take the time to catch up on earlier assignments, ask questions about the things they didn't yet understand, get an early start on the summative assessment, and spend more of their mental bandwidth on the end-of-term reflection assignment.

If I were teaching the course again, I would shift Assignment 9 a week earlier and get rid of Assignment 8 altogether. This would give students an opportunity to implement part of the Fourier transform weeks before they have to try it for the summative assessment. I found that students had conceptual issues with the Fourier transform approximation that were unrelated to the structure of the code and didn't emerge until students were already working on the summative assessment; if they had had a chance to practice earlier, their questions could have been resolved earlier. Although Assignment 8 was relevant to the course content overall, it contributed the least to the scaffolding leading up to the summative assessment.