Grading at Scale in EarSketch

Avneesh Sarwate
Georgia Tech Center for Music Technology
Atlanta, Georgia
avneesh@gatech.edu

Creston Brunch
Georgia Tech Center for Music Technology
Atlanta, Georgia
crestonbrunch@gmail.com

Jason Freeman
Georgia Tech Center for Music Technology
Atlanta, Georgia
jason.freeman@gatech.edu

Sebastian Siva
Georgia Gwinnett College
Lawrenceville, Georgia
ssiva@ggc.edu

ABSTRACT
This paper explores some of the challenges posed by automated grading of programming assignments in a STEAM (Science, Technology, Engineering, Art, and Math) based curriculum, as well as how these challenges are addressed in the automatic grading processes used in EarSketch, a music-based educational programming environment developed at Georgia Tech. This work-in-progress paper reviews common strategies for grading programming assignments at scale and discusses how they are combined in EarSketch to evaluate open ended STEAM-focused assignments.

Author Keywords
Education; STEAM; Web Audio; Automatic Grading; MOOC;

ACM Classification Keywords
K.3 EDUCATION; J.5 ARTS AND HUMANITIES

INTRODUCTION
Automatic grading has long been investigated in computer science pedagogy as an avenue for improving the quality of programming education [9]. In recent years, with the explosion of enrollment in computer science classes [4], automatic grading tools are more important than ever as instructors try to provide meaningful feedback on assignments with an ever increasing number of students. Auto-graders are also essential to the growing number of computer science classes and degree programs that are offered in a MOOC format [5].

At the same time that computer science enrollments have grown, there has been a movement to increase the role of expressive, creative learning contexts for introductory computer science courses that combine computing and creativity in a STEAM (STEM + arts) context [10]. A growing body of research suggests the potential of such learning contexts to broaden participation and engagement in computing [8, 14, 12].

Many of the popular frameworks for autograding primarily utilize grading methods based on unit tests to check for certain program outputs and sometimes static analysis to check for certain code structures [2, 6]. And although these tools are quite mature, there are still some unique challenges for effectively autograding STEAM assignments for which these strategies fall short. STEAM curricula tend to emphasize assignments that allow for a much more open ended scope with regard to the final product of the assignment, and this poses a fundamental challenge for automated grading. The test-passing or structure-matching approaches usually taken towards automatic grading are difficult to deploy when the end goal of an assignment has no explicit "ground truth" to compare against.

EarSketch is a computer science learning environment in which students algorithmically construct songs via code [10]. It has been used primarily at the high school level for Computer Science Principles courses [1] and in the college level for CS0 courses, but has also been used within a MOOC[1]. As a programming tool deployed in a STEAM curriculum, assignments in EarSketch often have the type of open-endedness that makes automatic grading challenging. No single auto-grading approach is sufficient to effectively assess such student projects at scale. Instead, EarSketch uses a mix of grading strategies that together comprehensively serve to evaluate students' work. In this paper, we summarize these various strategies and their application to different types of student assignments within EarSketch and, more generally, in STEAM computing learning contexts.

This work-in-progress paper reviews some common strategies for the automated grading of programming assignments, as well as discussing the current state of automated grading in EarSketch. The currently implemented strategies are part of a larger investigation into code pattern analysis and peer-grading as they relate to STEAM based computer science education.

POPULAR TOOLS AND COMMON STRATEGIES
Though a thorough survey of autograding platforms is beyond the scope of this paper, recent works [2, 5] have compared the features of several mature frameworks. The

L@S 2018, June 26–28, 2018, London, United Kingdom
© 2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5886-6/18/06$15.00
https://doi.org/10.1145/3231644.3231708

[1] https://www.coursera.org/learn/music-technology

analyzed frameworks shared several common strategies towards automatic grading.

Many of the frameworks utilize unit testing to validate programs, making sure that specific inputs to student programs result in specific outputs. These unit tests are run either as a shell script that parses the output of the programs, or by defining the assignment with an API and compiling and running the student program as part of a larger testing suite.

Some frameworks extend this unit testing paradigm. Marmoset [15] combines public and hidden unit tests with a release methodology that encourages students to submit assignments early and iteratively refine them. Web-cat [7] and Marmoset can also both use student created tests as part of the grading metric by analyzing the code coverage from the student created tests.

Static analysis is another common strategy in automated grading. Static analysis has a long history in automated debugging [17]. One popular Java static debugger, FindBugs [3], is used in Marmoset to grade programs. Also, many platforms support the writing of arbitrary shell scripts as grading programs, enabling instructors to integrate a static analyzer of their choice into the grading pipeline. In AutoLep [18], instructors can upload several reference solutions for each assignment. The ASTs of the student assignments are then compared against the ASTs of the set of reference assignments to check for a match. Assignments that cannot be verified as correct against the ASTs of reference solutions are further graded via more traditional test case approaches. The okpy framework [16] uses static analysis to assist students as they write code: it snapshots incremental progress of individual student assignments and integrates "automatic composition feedback, targeted conceptual hints, and dynamically generated code fixes into the OK system" [16].

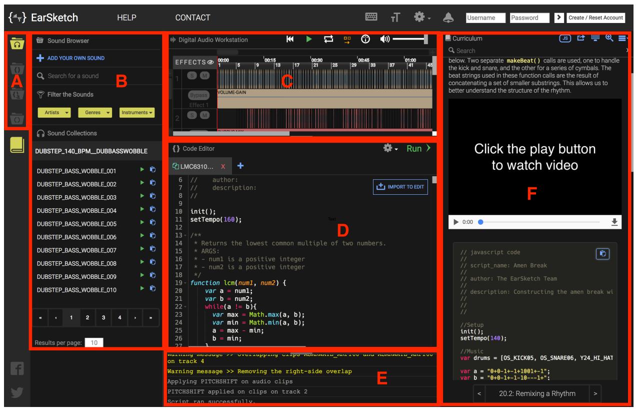
EARSKETCH

EarSketch [11] is a web based programming environment that enables students to write and run JavaScript or Python code to generate music.

Students use EarSketch to programmatically sequence and arrange premade audio clips, create step-sequenced rhythms, and manipulate audio effects across multiple tracks of a music production environment.

Figure 1 highlights the main interface components of the EarSketch environment, which include a sound library browser (B), a code editor (D), a multi-track digital audio workstation view that shows the musical results of code execution (C), and an inline curriculum browser (F).

Internally, the results of script execution are a JSON object that describes the time-placement of audio clips and effects on each track. It is this structure, along with the source code, that can be used in autograding assignments.



```
//    author:
//    description:
//

init();
setTempo(160);

/**
 * Returns the lowest common multiple of two numbers.
 * ARGS:
 * - num1 is a positive integer
 * - num2 is a positive integer
 */
function lcm(num1, num2) {
    var a = num1;
    var b = num2;
    while(a != b){
      var max = Math.max(a, b);
      var min = Math.min(a, b);
      a = max - min;
      b = min;
```

```
on track 4
Warning message >> Removing the right-side overlap
Applying PITCHSHIFT on audio clips
PITCHSHIFT applied on clips on track 2
Script ran successfully.
```

below. Two separate makeBeat() calls are used, one to handle the kick and snare, and the other for a series of cymbals. The beat strings used in these function calls are the result of concatenating a set of smaller substrings. This allows us to better understand the structure of the rhythm.

```
// javascript code
//
// script_name: Amen Break
//
// author: The EarSketch Team
//
// description: Constructing the amen break wi
//
//
//

//Setup
init();
setTempo(140);

//Music
var drums = [OS_KICK05, OS_SNARE06, Y24_HI_HAT

var a = "0+0-1+-1+1001+-1";
var b = "0+0-1+-1-10---1+";
```

Figure 1. The EarSketch Environment

AUTOGRADING IN EARSKETCH

EarSketch employs two different automatic grading tools for evaluating assignments.

The first module (Autograder-1) is a more traditional unit-test style system, while the second (Autograder-2), is an in-progress platform to allow for more flexible grading of songs developed in EarSketch.

Autograder-1

The Autograder-1 module allows instructors to grade student assignments against a reference solution. The web interface allows an instructor to compare the output of student programs against that of a reference program selected by an instructor. If the reference script requires user input, a prompt will appear asking for the value, and the same value will be autofilled for all input prompts from student scripts. A particular static random seed will be used for all random number generator calls in both reference and student scripts to decrease the likelihood that randomness will cause differences in output between scripts. After the scripts are all run, their results are compared against the reference, and the non-matching assignments are highlighted. The tabs for assignments with non-matching results can be expanded so the instructor can look at the code and multi-track music representation for that assignment.

The EarSketch CS0 curriculum[2] contains many micro-assignments throughout the chapter. These micro-assignments have tight specifications, such as changing a given script such that it produces a particular musical output. While these micro-assignments are effective at providing opportunities for students to practice key computational concepts (e.g. loops, conditionals) under clearly defined constraints, they allow little room for student creativity and self expression, since there is always a single correct musical solution to an assignment. We initially tried to address this by asking students to add their own musical ideas in addition to the prescribed musical

[2] http://earsketch.gatech.edu/itec2120/book.html

output, and then only auto-grade certain tracks and/or measures of music. But this still highly constrained the creative possibilities for musical output, and it became clear that within an expressive STEAM computing environment, students needed to complete more open-ended assignments to complement these micro-assignments in order to be fully engaged. An example of such end-of-module assignments would be to create a song that prompts the user for text input and then uses an if/else block to evaluate that input and generate the music accordingly. This kind of assignment necessitated the development of a new approach to auto-grading.

Autograder-2

The Autograder-2 module uses both static and dynamic testing approaches to tackle the challenge of grading of "open ended" assignments, such as creating introductory music for a radio show that evokes a particular mood or transforming a data set into music. It provides an API for instructors to write their own Python scripts for grading EarSketch scripts. Like Autograder-1, it provides functions for simulating user inputs. The two key components of the Autograder-2 API are the complexity scorer and the result object retriever.

The complexity scorer analyzes the AST of an EarSketch script and returns a "complexity report", which counts the number of usages of particular syntactic structures used in the script. In particular, the syntactic structures counted are user defined functions, conditional statements, boolean logic statements, loops, lists, list manipulation operations, and string manipulation operations. The built in complexity analyzer in EarSketch [figure?] provides an example of how these component scores can be used to generate a single numeric complexity score for a script. The complexity API allows instructors to create their own weightings for the task at hand (e.g - in a homework focusing on modularity of code, the scoring function could have a higher weighting for the number of functions used).

The API also allows instructors to access the "result" object, which is the JSON object that holds information on the sequencing and arrangement of audio clips and effects in the final rendered song. Instructors can use this info to enforce certain restrictions on the minimum number of audio items that must be used in a track, or song duration, or numbers of effects. For example, an assignment could have the restriction that a song must have at least 4 tracks and at least 10 different audio clips.

These kinds of "complexity floors" are a way to quantify grading without explicitly dictating what must be created, but they still suffer from some shortcomings. First, we found that students often implicitly equate a higher complexity with a higher score, when in practice only a fixed number of points are given for achieving a particular benchmarks (e.g, 5 points for defining 3 or more functions). However, since more complex code does not necessarily equate to better code, creating scoring systems that reward arbitrarily complex code does not align with learning objectives in an introductory computing course. Also, it can be relatively easy to "game" the scoring system. Students could simply define unused functions, conditionals, or loops to increase their code score without meaningfully using these programming constructs. In an effort to reduce student incentives towards writing needlessly complex code, the label for the built-in complexity analyzer was renamed from "Code Complexity" to "Code Concept Indicator," and curriculum assignments make explicit that there is no reward for exceeding the "floor" requirement for a computing construct.

PEER GRADING

Before the development of Autograder-2, peer grading was used in the EarSketch MOOC course to accomplish many similar grading tasks at scale. Students were asked to evaluate peer EarSketch scripts against a rubric that checked if the script met the requirements of an open-ended assignment in terms of musical content (i.e. tracks, sounds, etc.) and computational content (i.e. lists, loops, function definitions). While this placed an additional burden on students and presented challenges with respect to rater reliability, it also afforded students the opportunity to learn from each other's projects and to offer feedback on the creative and musical aspects of the work that no autograder can currently provide.

It was with this final goal in mind that we more recently designed a peer-grading system that complements the EarSketch Autograders, supporting a studio based learning [13] approach to student project review and feedback. After students have submitted their projects, they review three other student submissions and a) verify that they satisfy project requirements and then b) rank them and offer feedback in terms of their musical characteristics. In class, the highest-ranked assignments are then presented and discussed. The purpose of these "design crits" is to provide both peer and instructor feedback, allowing students to hone both their design and communication skills. The peer grading process is facilitated by a bespoke learning management system plugin that integrates with the share-by-URL feature of the EarSketch learning environment.

CONCLUSION AND FUTURE WORK

EarSketch uses a variety of strategies to provide a scalable grading framework in a STEAM curriculum. The curriculum's mix of strictly scoped and open ended assignments allows for both focused reinforcement of specific skills as well as freer exploration of coding and music. The variety of assignment types and grading strategies both allows particular automated grading strategies to be applied only to assignment types for which they are appropriate, and ensures that all assignment types have some form of scalable grading procedure. Furthermore, the integration of automatic grading and studio based learning can give instructors a powerful set of tools for analyzing assignments for trends prior to a studio session. Moving forwards, we hope to incorporate more sophisticated pattern recognition methods (such as deep learning) into our automated grading procedures and to further incorporate automated grading and studio-style crits into a holistic scalable feedback process.

ACKNOWLEDGMENTS
EarSketch receives funding from the National Science Foundation (CNS #1138469, DRL #1417835, DUE #1504293, DRL #1612644, and IIP #1741045), the Scott Hudgens Family Foundation, the Arthur M. Blank Family Foundation, the Ruth L. Seigel Family Foundation, and the Google Inc. Fund of Tides Foundation. EarSketch is available online at earsketch.gatech.edu.

Any opinions, findings, and conclusions or recommendations expressed in these materials are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding bodies.

REFERENCES

1. Astrachan, Owen, and Amy Briggs. "The CS principles project." *ACM Inroads* 3, no. 2 (2012): 38-42.
2. Caiza, Julio C., and José María del Álamo Ramiro. "Programming assignments automatic grading: review of tools and implementations." (2013): 5691-5700.
3. Cole, Brian, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. "Improving your software using static analysis to find bugs." In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 673-674. ACM, 2006.
4. Computing Research Association. "Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006. 2017." (2017).
5. Daradoumis, Thanasis, Roxana Bassi, Fatos Xhafa, and Santi Caballé. "A review on massive e-learning (MOOC) design, delivery and assessment." In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pp. 208-213. IEEE, 2013.
6. DeNero, John, Sumukh Sridhara, Manuel Pérez-Quiñones, Aatish Nayak, and Ben Leong. "Beyond Autograding: Advances in Student Feedback Platforms." In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 651-652. ACM, 2017.
7. Edwards, S. H., and Perez-Quinones, M. A. Web-cat: automatically grading programming assignments. In ACM SIGCSE Bulletin, vol. 40, ACM (2008), 328–328.
8. Guzdial, Mark. "A media computation course for non-majors." In *ACM SIGCSE Bulletin*, vol. 35, no. 3, pp. 104-108. ACM, 2003.
9. Hext, Jan B., and J. W. Winings. "An automatic grading scheme for simple programming exercises." *Communications of the ACM* 12, no. 5 (1969): 272-275.
10. Maeda, John "STEM + Art = STEAM," *The STEAM Journal*: Vol. 1: Iss. 1, Article 34. (2013)
11. NEWCITE Magerko, Brian, Jason Freeman, Tom Mcklin, Mike Reilly, Elise Livingston, Scott Mccoid, and Andrea Crews-Brown. "Earsketch: A steam-based approach for underrepresented populations in high school computer science education." *ACM Transactions on Computing Education (TOCE)* 16, no. 4 (2016): 14.
12. McCoid, Scott, Jason Freeman, Brian Magerko, Christopher Michaud, Tom Jenkins, Tom Mcklin, and Hera Kan. "EarSketch: An integrated approach to teaching introductory computer music." *Organised Sound* 18, no. 2 (2013): 146-160.
13. Narayanan, N. Hari, Christopher Hundhausen, Dean Hendrix, and Martha Crosby. "Transforming the CS classroom with studio-based learning." In Proceedings of the 43rd ACM technical symposium on Computer Science Education, pp. 165-166. ACM, 2012.
14. Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner et al. "Scratch: programming for all." *Communications of the ACM* 52, no. 11 (2009): 60-67.
15. Spacco, Jaime, David Hovemeyer, and William Pugh. "An Eclipse-based course project snapshot and submission system." In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pp. 52-56. ACM, 2004.
16. Sumukh Sridhara, Brian Hou, Jeffrey Lu, and John DeNero. 2016. Fuzz Testing Projects in Massive Courses. In Proceedings of the Third (2016) ACM Conference on Learning @ Scale (L@S '16). ACM, New York, NY, USA, 361-367.
17. Tischler, Ron, Robin Schaufler, and Charlotte Payne. "Static analysis of programs as an aid to debugging." In *ACM SIGPLAN Notices*, vol. 18, no. 8, pp. 155-158. ACM, 1983.
18. Wang, T., Su, X., Ma, P., Wang, Y., Wang, K. Ability-training-oriented Automated Assessment in Introductory Programming Course. Computer. Education, Elsevier, vol. 56, pp. 220-226. 2011