Literature Review of Computational Thinking, Computing for Life Sciences, and Graduate Students in Statistics

Allison Theobold

The intention of a review of the literature is (1) to situate the importance of computing in the sciences; (2) to understand what is known about computational abilities of undergraduate and graduate students in life science fields, and (3) what is known about graduate students in statistics courses.

Importance of Computing in Mathematics and the Sciences

Over the last decade the life science fields have seen a rapid increase in the use of computation and analytical tools to model phenomena across many disciplines of inquiry. In some scientific fields, such as biology and chemistry, the recent ability to collect multitudes of data easily and quickly have made computational abilities vital to researchers and practitioners. Meanwhile, fields previously thought to be niche disciplines, such as computational and mathematical biology, are now "becoming an integral part of the practice of biology across all fields (Stefan, Gutlerner, Born, & Springer, 2015). Across a large sector of scientific domains applications of mathematical and statistical techniques, such as Markov Chain Monte Carlo, neural networks, and agglomerative hierarchical clustering, have become essential computational understandings in field applications (Weintrop et al., 2016). With these advances both in data collection, visualization, analysis, and interpretation as well as computational power, analytical methods, and detailed computational models, scientific fields are undergoing a renaissance. These advances have, however, created a growing need for life scientists to receive an appropriate education in computational methods and techniques. The need for computation in education for mathematics and science is greater than ever (Fox & Ouellette, 2013).

Computing in the Life Sciences

Research in computational abilities of Applied Biological Science majors is in its infancy, with only a handful of institutions performing research that specifically addresses the computation training necessary to prepare students for careers post undergraduate or graduate school. In this section we discuss briefly three literatures that have informed this study. We will discuss the research efforts on curriculum design for introductory computing courses for non-computer science students. Finally, we describe one institution's development and implementation of computational training for graduate students in the biological sciences.

Computational Courses for Undergraduate Science Majors

Multi-disciplinary efforts have been made at Purdue, Carnegie-Mellon, Harvey Mudd, Princeton, and University of Toronto (Cortina, 2007, Dodds, Alvarado, Kuenning, & Libeskind-Hadas (2007), Dodds, Libeskind-Hadas, Alvarado, & Kuenning (2008), R. Sedgewich & Wayne (2008), R. Sedgewich & Wayne (2015), Wilson, Alvarado, Campbell, Landau, & Sedgewich (2008), Wing (2006), Hambrusch, Hoffmann, Korb, Haugan, & Hosking (2009)) to create introductory computing courses with a focus on non-computer science majors, in particular science students, for fields such as physics, engineering, mathematics, chemistry, and biology.

Harvey Mudd

The *CS 1 for Scientists* course developed at Harvey Mudd (Dodds et al., 2007) was designed to provide a one semester overview of the discipline of computer science for future scientists, without the presumption that students would continue with computer science after the course. This course satisfies the institutional requirement that all students take a single introductory computer science course in the fall of their freshman year (Wilson et al., 2008). Additionally, the computer science faculty at Harvey Mudd strive for the course to remain both current and relevant. The course does not emphasize any specific tools, like object oriented programming, instead striving to paint a broader picture of the skills vital to all areas of science. The faculty's goals for the course were to (1) develop programming and problem solving skills useful across all areas of science, (2) attract students to continue their computer science education, and (3) provide an approachable and compelling picture of computer science (Dodds et al., 2007).

The curriculum developed for *CS 1 for Scientists* begins with a functional approach, "building upon students' prior ability and comfort with mathematical functions" (Dodds et al., 2007, p. 24). Python is the language used for the course, as it "is simple enough for an introductory course, versatile enough to illustrate a broad range of programming paradigms, and powerful enough to be used by scientists in many disciplines" (Wilson et al., 2008, p. 36). For the second unit of the semester, students progress through logic gates using AND, NOT, and OR structures, using Carl Burch's Logisim tool (Burch, 2002). Students then progress through imperative programming, using Conway's Game of Life, class and object based constructs, using date structures, and finally an "bird's-eye view of what computing can and cannot do," (Dodds et al., 2007, p. 25) an overview of (un)computability.

To assess the course's success on conveying both the importance and the breadth of computer science, the department implemented pre- and post-term assessments. Instructors asked students "What is computer science?" and "Describe one thing a researcher in computer science might study." (Dodds et al., 2007, p. 26). Students' responses were categorized into four themes: none, naive, basics, and details. In the none theme, students provided non-answers, such as "the science of computers" or "using code to get computers to do things" (Dodds et al., 2007, p. 26). The naive theme contained student responses which conveyed a breadth within computer science, such as "software and hardware design" and "coding, debugging, and analyzing problems to develop computer-based solutions" (Dodds et al., 2007, p. 26). The basics theme includes student responses that go beyond physical computers and their software. Finally, in the details them students provided "the most nuanced" responses, such as "a lot is about general, language-agnostic even system-agnostic algorithms and relative merits of speed and efficiency and in some cases... actually wondering how and if it is possible. CS is not programming, it is implemented in programming" (Dodds et al., 2007, p. 26).

Harvey Mudd has seen that approximately 85% of the students who enroll in this course will go on to pursue a degree in a scientific discipline other than computer science (Wilson et al., 2008). In comparison with the "traditional" CS 1, *CS 1 for Scientists* saw an increase in students' overall stimulation of interest, as well as their computer science self-efficacy. Instructors of this course feel that "compared to students completing the previous offering of the course, students completing this course have a better understanding of the field of computer science, [and] are equally proficient or superior programmers, and have a better understanding of the relationship of computer science to other scientific disciplines" (Wilson et al., 2008, pp. 36–37).

Purdue

At Purdue, all science undergraduates must fulfill a computing requirement and all science majors have a multidisciplinary requirement for their degree (University, 2007). The new *Introduction to Computational Thinking* course developed at Purdue was a collaborative effort between faculty in Computer Science, Physics, Biology, Chemistry, and Statistics. This course is intended to begin each students journey into computing, taking a problem-driven approach and focusing on learning computational methods through scientific discovery. Students are presented examples in a familiar language that allows for them to focus on the foundational principles of each computational problem (Hambrusch et al., 2009). Only the programming features that have a direct relationship with problems the students are solving are presented, eliminating route memorization of

techniques without application specific understanding. In using Python, the goal of the faculty is that students are able to write meaningful programs in a short amount of time, without the burden of language details. Finally, a new different emphasis than the courses developed at Harvey Mudd and Princeton, visualization is emphasized for its ability to bring scientific findings to life.

In constructing these five principles of the course, the Physics, Chemistry, and Bioinformatics departments conveyed their expectations on what they wished their students would learn through this course. The Physics faculty wished for students to gain a more complete understanding of computing and a perspective on the interplay between physics, applied mathematics, and computer science, in solving problems computationally. The computational Chemistry faculty were interested in students learning computational methods relevant to their field, such as Monte Carlo, Simulated Annealing, and Molecular Dynamics. Bioinformatics faculty communicated the use of `R` in their field for both statistical computing and visualization. Faculty in Computer Science value these interactions with other science faculty. These conversations have the ability to shed light on how the material taught in this course relates to material students will see in later science courses.

Following these discussions of faculty expectations, a 15-week course, with two one-hour lectures and two one-hour labs per week was developed (University, 2008). Python was chosen as the programming language due to its "interactive environment, ability to let novice programmers quickly write non-trivial programs, adoption by many scientific communities, and support for numerous specialist libraries" (Hambrusch et al., 2009, p. 185). The course did delve into the basics of object-oriented design near the end of the semester, where it felt quite natural for students to comprehend. The use of visualization helped students to better comprehend the scientific questions being asked in the course projects.

The first offering of this course was in spring 2008, with a total of 13 students from Physics and Chemistry. In a comparison of the entry and exit responses of students to the questions, "How would you rate your current interest in":

- Taking another computer science course?
- Pursuing a career that requires programming skills? (Hambrusch et al., 2009)

researchers found substantial increases in student interest in computer science. With the success of this first course, computer science faculty planned to build a second course in which "students collaborate in teams that include both science and CS majors" (Hambrusch et al., 2009, p. 187). This course will further solidify the importance of collaborations between computer science and other sciences, as well as stress the importance of working in collaborative teams.

Carnegie-Mellon

In the fall of 2005 Carnegie Mellon began to teach a new course titled *Principles of Computation* is geared towards non-majors in non-technical fields (majors that are not solely mathematical, scientific, or engineering motivated) (Cortina, 2007). The course was motivated by students only enrolling in introductory CS courses out of a requirement and were not expected to use programming skills in their chosen field. *Principles of Computation* introduces students to the principles of computation, how computer scientists use computation through algorithms, algorithm efficiency and correctness, the limits of computation, and unique applications of computer science in the real world. Students are not expected to program in the course, instead using a flowchart simulator to computationally reason with algorithms and the process of computing. With this deemphasis on programming, students are better able to "understand the power of computing and the unique problems we face in computer science that can affect their disciplines" (Cortina, 2007, p. 218). The computer science faculty hope that this course "will allow the department to help spread the word about the world of computer science to students who have a biased perspective due to the overemphasis of learning a programming language as the first introduction to computer science" (Cortina, 2007, p. 220).

In this course, computational reasoning through algorithms is facilitated by Raptor (Carlisle & Handfield, 2005), a flowchart simulation tool. Raptor supports conditional statements, loops, subroutines, arrays, and graphics, and eliminates the syntax overhead in implementing an algorithm. Additionally, Raptor provides a

debugging tool, allows for the user to control the speed of the simulation, and allows users to trace variables during a simulation to see what the algorithm is doing.

Surveys administered at the close of the course showed that the majority (55%) of students are interested in taking another CS course following *Principles of Computation* and 85% of students would recommend this course to their peers. The researchers have further plans to offer the course as an accelerated study summer course for high school students, with college credit given for successful completion.

University of Toronto

In 2008, the University of Toronto had recently introduced a first-year computer science course aimed at science students. This course includes teaching a sufficient amount of programming and technical material to show students the practical value of computer science, while also introducing computer science to a larger audience of science students. This course can give science students "a path into, and possibly attract them to [computer science]" (Wilson et al., 2008, p. 37). Unlike the courses at Harvey Mudd and Purdue, the course at University of Toronto introduces atypical topics, such as databases and 2D plotting. The university has also developed a follow-up course for interested students, so they can enter second year courses with the same number of CS courses as a CS major.

Princeton

Sedgewick, in discussing the development of the standard mathematics and science curriculum prior to the advent of computing, laments "two unfortunate developments at the dawn of the new millennium." First, computer science students "have somehow been exempted" from learning general scientific precepts and mathematics. Second, many students in the sciences, mathematics, and engineering "have somehow been exempted" from learning the foundational concepts of computer science, instead learning only tools relevant to their studies. "In both cases, students are being shortchanged" (Wilson et al., 2008, p. 37). He then argues that computer science needs to be integrated into the standard curriculum, so that *all* students interested in science and engineering develop foundational understandings of mathematics, science, and computing before moving toward their discipline-specific studies.

Conclusions

The research on undergraduate level computing courses directly relates to the computational abilities of graduate level science researchers, as these students have most likely graduated from an undergraduate life science program with no computational training. The concepts emphasized in the courses developed at Purdue, and elsewhere, can be used to inform Applied Biological Science and Statistics departments as to what computational skills other scientific disciplines, such as physics, biology, and chemistry, believe to be the most important for students to grasp.

Computational Training for Graduate Science Majors

Workshops for Bioinformatics Graduate Students

Researchers in the Department of Biological and Biomedical Sciences at Harvard have developed an intensive course that introduced graduate students to the "fundamentals of programming, statistics, and image and data analysis through the use of MATLAB." (Stefan et al., 2015) These courses are framed not only with the goal of students developing programming skills, but also emphasizing students learning how to algorithmically reason through a computational problem. The structure of the 50-hour course dedicates the first two days

to an introduction to programming using MATLAB, where students learn a variety of topics, including creating variables, performing basic variable operations, indexing, logicals, functions, conditionals, and loops. These courses are given twice a year, once prior to the start of the school year as new graduate students are attending orientation, and a second time for "students who realize the need for such training later in their studies."(Gutlerner & Van Vactor, 2013) In introducing beginning graduate students to these concepts, researchers hope to lower the computational barrier for students taking courses, empower student to learn computational tools on their own, as well as allow other courses to "build upon this foundation and integrate quantitative methods throughout the curriculum" (Stefan et al., 2015).

`R` Labs and Lectures for Graduate (and Undergraduate) Computational Biology Students

Eglen, from the the Computational Biology Institute at the University of Cambridge, discusses his experiences in teaching `R` to computational biology graduate students in (Eglen, 2009). The institute had thought previously that students would self-learn `R`, however students communicated that `R` was too difficult to learn on top of all of their coursework. In response to this feedback, the institution developed a set of lectures and lab sessions that cover an introduction to programming in `R`. Similar to graduate level courses in Statistics, the faculty noticed that students come from a variety of backgrounds, some with programming experience and others without any programming experience. The lecture material they provide students contains concepts that novice programmers may not understand immediately, but will be a reference tool for them later in their coursework. Additionally, in developing the lecture notes, faculty aimed to emphasize the use of `R` as a general programming language, not a language that is specific to solving computational biology problems.

These labs include tutorials on the use of Sweave documents, incorporating a separate lab session on LaTeX. Students in the Computational Biology Institute are asked to submit their coursework in the form of Sweave documents, further emphasizing the importance of reproducible research. Throughout the implementation of the labs, Eglen has noticed common problems students have when learning `R`, which include syntax, knowledge of inbuilt functions, where to seek help, variable allocation, vectorization (optimization) techniques, and when to use different types of data.

Bioinformatics Workshops Impact on Research and Careers

Bioinformatics.ca has hosted continuing education programs for topics in both introductory and advanced bioinformatics in Canada since 1999. These workshops have impacted over 2,000 participants by June of 2016, but no research had been done on the long-term impact of these workshops on participants careers or research outcomes. The workshops offered by bioinformatics.ca are between 2 days and 5 days with a maximum of 30 students each (Brazas & Ouellette, 2016). The facilitation of all workshops begins with either pre-workshop online exercises and/or articles selected to give participants background preparation. During the workshop, faculty provide "brief lectures on the data analysis concepts and the bioinformatic approach as well as give an overview of the relevant bioinformatic tools" (Brazas & Ouellette, 2016, p. 2). These concepts are then demonstrated in a hands-on environment with a provided data set. Following this guided analysis, participants are given the opportunity to practice the analysis on a secondary data set or use their own data, when applicable. Participants' skill development is never formally measured during the workshop and retention or usage beyond the workshop has never been assessed.

Researchers from the University of Toronto and the Ontario Institute for Cancer Research emailed 1,276 of the bioinformatics.ca workshop participants, from the years 1999 to 2013. Of the participants contacted, 267 persons responded to all questions regarding how the workshops impacted their careers and/or their research. Of the respondents, "128 respondents felt the workshops helped them communicate better with bioinformaticians and statisticians" (Brazas & Ouellette, 2016, p. 6), 133 participants felt that they conducted better research following their participation in the workshop, 91 respondents stated that they used the analyses they learned in the workshop(s) to validate their results, and 76 participants used the skills they learned in the workshops for research publications.

Based on their experiences evaluating the impacts of these workshops, researchers would recommend evaluating skill retention and usage at approximately 6 months post workshop, on an individual workshop basis. As time since workshop could play a large role in the impact of these workshops, more timely evaluations should be implemented. Additionally, researchers would recommend publication, and research and career progression evaluations at 3-5 years post workshop. Lastly, due to the smaller response rate or 21%, these researchers would recommend requesting a permanent email address, for ease of tracking participants post workshop.

Conclusions

Graduate level terminal statistics courses, such as Methods of Data Analysis I and II, are taken by graduate Applied Biological Science students across the country, potentially acting as the final computational training students receive prior to performing independent research. These such courses provide a natural extension of the research on computational training and thinking of graduate students in the biological sciences, as both provide the computational training used by graduate students in their independent research. Applied Biological Science students have similar computational burdens to Biology students, with computational expectations often being placed on them with little to no training.

Computational Thinking

Efforts in Understanding Computational Thinking

Research in computational thinking for science majors is in its infancy, with only a handful of institutions performing research that specifically addresses how to teach and how students learn computational thinking. Primarily, the research being performed is implemented at the undergraduate level, with no research efforts made in the realm of graduate science students. Due to this gap in the literature, the reviewed literature were selected from research on computational thinking and abilities of undergraduate science students

The most substantial efforts being made in understanding computational thinking are by Harvard, where researchers are building tools for assessing the development of computational thinking. They have "relied primarily on three approaches: (1) artifact-based interviews, (2) design scenarios, and (3) learner documentation." [**?**] These scenarios were developed in collaboration with researchers at the Education Development Center and are used in the context of Scratch, a computer programming environment for use by elementary through high school students, created by the Lifelong Kindergarten Group at MIT Media Lab. The researchers at Harvard developed three different Scratch projects of increasing complexity, and in a series of interviews "students were presented with the design scenarios, which were framed as projects that were created by another young Scratcher. The students were then asked to select one of the projects from each set, and (1) explain what the selected project does, (2) describe how it could be extended, (3) fix a bug, and (4) remix the project by adding a feature." [**?**] Through these interviews, substantial contributions have been made in how to assess the ways students engage in computational thinking, with the ability to apply the structure of these interviews to other types of students and other types of computational thinking scenarios.

Graduate Students in Statistics

References

Brazas, M. D., & Ouellette, B. F. F. (2016). Continuing Education Workshops in Bioinformatics Positively Impact Research and Careers. *PLOS Computational Biology*, *12*(6), 1–12.

Burch, C. (2002). Logisim: A graphical system for logic circuit design and simulation. *Journal on Educational Resources in Computing (JERIC)*, *2*(1), 5–16.

Carlisle, W., M., & Handfield, S. (2005). RAPTOR: A visual programming environment for teaching algorithmic problem solving. In *Proceedings of the 36th sigcse technical symposium with computer science education* (pp. 176–180). ACM.

Cortina, T. (2007). An introduction to computer science for non-majors using principles of computation. In *Proceedings of the 38th sigcse technical symposium with computer science education* (Vol. 39, pp. 218–222). ACM.

Dodds, Z., Alvarado, C., Kuenning, G., & Libeskind-Hadas, R. (2007). Breadth-first cs 1 for scientists. In *Proceedings of the 12th annual sigcse conference on innovation and technology in computer science education* (pp. 23–27). New York, NY, USA: ACM.

Dodds, Z., Libeskind-Hadas, R., Alvarado, C., & Kuenning, G. (2008). Evaluating a breadth-first CS 1 for scientists. In *Proceedings of the 39th sigcse technical symposium with computer science education* (pp. 266–270). Portland, OR: ACM.

Eglen, S. (2009). A Quick Guide to Teaching R Programming to Computational Biology Students. *PLoS Computational Biology*, *5*(8), 1–4.

Fox, J. A., & Ouellette, B. F. F. (2013). Education in Computational Biology Today and Tomorrow. *PLOS Computational Biology*, *9*(12), 1–2.

Gutlerner, J. L., & Van Vactor, D. (2013). Catalyzing curriculum evolution in graduate science education. *Cell*, *153*(4), 731–736.

Hambrusch, S., Hoffmann, C., Korb, J., Haugan, M., & Hosking, A. (2009). A multidisciplinary approach towards computational thinking for science majors. In *Proceedings of the 40th sigcse technical symposium with computer science education* (Vol. 41, pp. 183–187).

Sedgewich, R., & Wayne, K. (2008). *Introduction to programming in java.* Addison Wesley.

Sedgewich, R., & Wayne, K. (2015). *Introduction to programming in python.* Addison Wesley.

Stefan, M. I., Gutlerner, J. L., Born, R. T., & Springer, M. (2015). The quantitative methods boot camp: Teaching quantitative thinking and computing skills to graduate students in the life sciences. *PLOS Computational Biology*, *11*(4), 1–12.

University, P. (2007). New science undegraduate curriculum. College of Science.

University, P. (2008). Lectures and course materials for "Introduction to coputational thinking". Computer Science. Retrieved from http://secant.cs.purdue.edu/cs190c:start

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, *25*(1), 127–147.

Wilson, G., Alvarado, C., Campbell, J., Landau, R., & Sedgewich, R. (2008). CS-1 for scientists. In *Proceedings of the 39th sigcse technical symposium with computer science education* (pp. 36–37). Portland, OR: ACM.

Wing, J. (2006). Computational thinking. *Communications of ACM*, *49*(3), 33–35.