ASEE 2022 ANNUAL CONFERENCE
Excellence Through Diversity
MINNEAPOLIS, MINNESOTA, JUNE 26TH-29TH, 2022 ASEE

Paper ID #37869

Fishers Handle Bugs Better than Fish-Receivers: Nourishing Computational Self-Efficacy in Engineering Coursework

Seyed-Arman Ghaffari-Zadeh (PhD student)

Arman Ghaffarizadeh is a Ph.D. student in Mechanical Engineering at Carnegie Mellon University. He obtained his Bachelor’s degree (high honours) and Master’s degree in Mechanical Engineering, from the University of Toronto in 2016 and 2018, respectively. He is a recipient of numerous NSERC awards for his undergraduate research (USRA), master’s work (CGS M), and PhD (CGS D/PGS D) studies. He was selected as a Russell A. Reynolds Graduate Fellow in thermodynamics and was awarded the Ontario Graduate Scholarship (OGS) for his work on multiphase systems. Arman is involved in numerous student committees and advocacy roles and was the recipient of the Society of Petroleum Engineers (SPE) and Jeremieh Mpagazehe Awards for his service and contributions to the graduate student community. In addition to his research focus on nanoscale transport phenomena, he conducted and published an international study on the effect of the COVID-19 pandemic on the life and work of academic researchers.

Gerald J. Wang (Assistant Professor)

Jerry Wang is an Assistant Professor of Civil and Environmental Engineering, and Mechanical Engineering (by courtesy) and Chemical Engineering (by courtesy), at Carnegie Mellon University. He received his BS in 2013 from Yale University (Mechanical Engineering, Mathematics and Physics), SM in 2015 from MIT (Mechanical Engineering), and PhD in 2019 from MIT (Mechanical Engineering and Computation). He performed postdoctoral research at MIT in Chemical Engineering. He was a member of the inaugural cohort of the Provost’s Inclusive Teaching Fellowship at CMU, was the 2020 recipient of the Frederick A. Howes Scholar Award in Computational Science and the 2016 MIT Graduate Teaching Award in the School of Engineering, and is an alumnus of the Department of Energy Computational Science Graduate Fellowship and the Tau Beta Pi Graduate Fellowship. Wang directs the Mechanics of Materials via Molecular and Multiscale Methods Laboratory (M5 Lab) at CMU, which focuses on computational micro- and nanoscale mechanics of fluids, soft matter, and active matter, with applications in Civil and Environmental Engineering across the nexus of water, energy, sustainable materials, and urban livability. The M5 Lab is particularly interested in particle-based simulations, systems out of equilibrium, uncertainty quantification in particle-based simulations, and high-performance computing. He teaches courses in molecular simulation and computational/data science.

Fishers Handle Bugs Better than Fish-Receivers: Nourishing Computational Self-Efficacy in Engineering Coursework

Abstract

As the use of computational tools and advanced computing principles proliferates in engineering practice, a growing number of engineering curricula place heavy emphasis on developing computational reasoning skills. This pedagogical imperative presents unique challenges in graduate-level computational engineering courses, which often feature enormous heterogeneities in undergraduate background, disciplinary training and interest, and — in particular — prior exposure to computer programming and associated best practices. In this work, we focus on a graduate-level course that features a series of progressively scaffolded assignments in which students develop an elaborate molecular simulation code. We present strategies that have been deployed in this course, aimed at encouraging the development of computational self-efficacy. We also provide qualitative and quantitative assessments of these strategies, with special attention dedicated to assessment techniques that foster a growth mindset in the context of improving computational efficiency. We explore how providing students with a chance of resubmitting assignments improves learning outcomes. The positive effects are especially pronounced for students who have a smaller number of prior computing-related courses. We also discuss trends observed as a function of students' preferred programming language, and correlations between preferred language (especially whether the language is compiled or interpreted) and likelihood of prioritizing computational efficiency. These results have natural implications for the inclusive and equitable growth of graduate-level computational engineering curricula, including and especially for graduate-level onboarding. Taken as a whole, our work highlights opportunities to encourage our students to — as the adage goes — seek substance beyond "proffered fish," learn how to "handle bugs," and eventually "fish for themselves."

Introduction

A growing number of mechanical engineering programs require at least one course in computational engineering at the undergraduate level, with many engineering departments now offering significant advanced undergraduate- and graduate-level computational engineering coursework (and, in many cases, graduate-level degrees or certificates in computational engineering). As the popularity of computational engineering continues to grow, especially at the MS and PhD levels, so too do the challenges of teaching courses in this field, given the wide range of prior experiences that students bring to their graduate-level coursework. To maintain diverse, equitable, and inclusive classrooms, it is critical to invest in and deploy pedagogical strategies that help less-experienced students thrive alongside their peers with more prior computing experience.

As has long been known to both teachers and students of computer programming, proficiency in computing requires significant iterative practice over a long period of time to develop both high-level conceptual proficiency in addition to "muscle memory" (see, e.g., [1], [2], for historical discussions of student development of computational problem-solving skills).[1] Unsurprisingly, there is considerable evidence that incremental, scaffolded exposure to programming concepts enhances student learning outcomes [3], [4]. Computational engineering brings an additional challenge in that it requires not only proficiency with computer programming, but also conceptual adeptness with an underlying body of domain-specific knowledge. Past work (see, e.g., [5]) has established that the opportunity to resubmit assignments in an undergraduate-level computing course improves student learning.

Following in the spirit of these prior results, in this work, we tackle the following question: "Does providing the opportunity to resubmit earlier assignments improve student learning outcomes in a graduate-level computational engineering course?" In the process of addressing this question, we also seek to answer a related question: "Are there patterns across students who use each computer language that might inform best practices for instruction in computational engineering?"

Educational Context and Methodology

This study was performed in the context of a course entitled, "Molecular Simulation of Materials" (MSM). This is a graduate-level course in the College of Engineering at Carnegie Mellon University. The typical enrollment of the class is between 15 and 25 students, primarily at the MS and PhD levels, with a small number (typically < 10%) of undergraduate students. The course has no official prerequisites, but assumes basic knowledge of calculus, linear algebra, probability and statistics, classical mechanics, and thermodynamics at the undergraduate level.

As one of the core deliverables in the course is the development of a relatively significant molecular simulation code from scratch (a task that requires on the order of 30-40 hours of effort), students are also expected to have basic proficiency with computer programming. The course staff flexibly accommodates students wishing to use any computer language amenable to scientific

[1] The same could, of course, be said for essentially any skill. Based upon the authors' anecdotal experience, this claim is especially true for the skill of computer programming. Moreover, computing is an area in which students tend to exhibit more significant heterogeneity as compared to, e.g., background with thermodynamics.

computing; in practice, all students have used one (or some combination) of C++, MATLAB, and Python. In a typical semester, there is a large amount of heterogeneity with regard to scientific computing background, ranging from less than one semester of prior experience to greater than five years of prior experience.

The course has six problem sets and a final project (written report and presentation) on a topic of the student's choosing. The (heavily scaffolded) problem sets guide students to develop a homegrown molecular simulation code that employs both the molecular dynamics (MD) and the Rosenbluth-Hastings-Metropolis Monte Carlo (MC) techniques. Each problem set contains a set of questions that emphasize theoretical principles (and generally do not require programming) as well as several problems that guide the student to continue developing their molecular simulation code.

Given the broad diversity of backgrounds with computing, students are not formally assessed or graded on the computational efficiency or scalability of their codes (students, of course, must develop computer programs that can run to completion by the assignment due dates in order to receive full credit). Provided that the relevant scientific principles are correctly implemented, full credit is awarded (in many cases, suggestions targeting greater efficiency are provided as feedback when assignments are returned). However, depending on the use of computationally efficient programming practices, the time it takes to complete each assignment can vary dramatically (in some cases by over two orders of magnitude, even when the underlying scientific principles are implemented entirely correctly). The ability to quickly identify and debug errors in code also depends upon being able to run the code itself (or at least small portions of it).

For problems that require computer programming, after receiving grades and feedback on their initial submission, students are allowed to "take a mulligan" (i.e., improve their code and resubmit). Full details on the resubmission policy are provided in the Appendix.

To gain insight into the questions identified in the previous section, we have compared the performance of students who consistently chose to take advantage of the resubmission policy against those who did not. We have also looked for common patterns between students who made use of Python vs. MATLAB, the two most heavily used languages in the semester this study was conducted.

Results

A total of 11 students took advantage of the resubmission policy, and all 11 students attained higher assignment scores. Nine of these 11 students showed substantial improvement, with score increases in excess of 25%. Students with limited prior programming experience featured prominently in this group. Students who took advantage of the resubmission policy had taken an average of 1.9 prior computing-related courses (Table 1), while students who did not resubmit had taken an average of 4.3 courses. The latter group scored significantly higher on their initial submissions.

	Prior computing-related courses[2]	Initial assignment grade
Single submission	4.3	99%
Resubmission	1.9	62%

Table 1: Averages for amount of prior computing-related coursework and initial assignment grade, split out by whether a student took advantage of the homework resubmission policy.

We noticed that the instructional staff played an important role in these improvements, as evidenced by interactions during office hours. Most of these students displayed engagement considerably beyond, "*my code does not work well, why is that?*", which we attribute to significant additional time to revisit earlier code (and in some cases completely carry out complete restructuring of their code). Some of these students even took a step further than having merely working code, and implemented additional features to accelerate their code, well beyond basic assignment requirements. We believe that this curiosity was motivated by their conversations with the instructional staff (and fundamentally enabled by the flexible resubmission policy).

Yet another observation (perhaps unsurprising to seasoned computational engineers) is that the prevalence of code vectorization (a consensus best practice that is associated with high computational efficiency) was higher amongst students electing to use MATLAB as compared to students electing to use Python (Figure 1). Although all students who did not initially use vectorization eventually performed some vectorized operations in subsequent problem set submissions, this (admittedly small-statistics) difference between Python and MATLAB users is intriguing, and potentially suggests a benefit to placing an emphasis at the undergraduate level on *any* language that naturally permits vectorization.

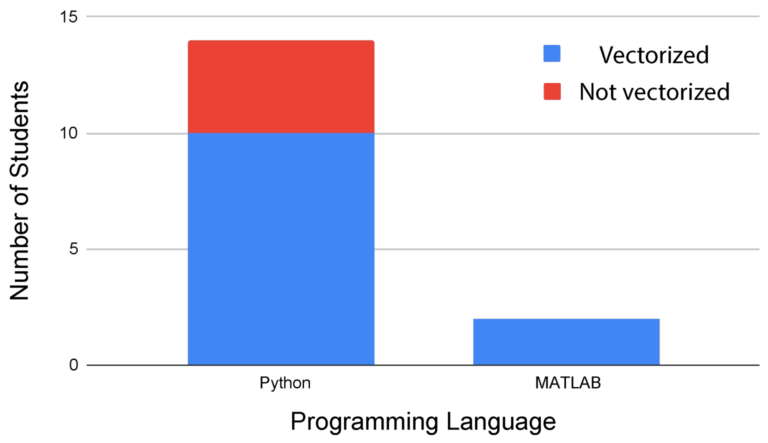

Figure 1: Number of students using each language, as well as a particular practice (vectorization) associated with computational efficiency.

[2] The term "computing-related courses" covers both computer science courses and computational science/engineering courses, either at an introductory or an advanced level (e.g., numerical methods, computational fluid dynamics, finite elements, algorithms, or databases).

From a course evaluation perspective, we have a sparse but encouraging body of results. This course has been offered twice by the current instructional team, with the policy on assignment resubmissions described in this work implemented in the second offering. Overall ratings of the course by mechanical engineering students increased from an average of 4.92/5.00 to 5.00/5.00 and ratings of the instructor increased from an average of 4.92/5.00 to 5.00/5.00. Although neither of these increases are, of course, statistically significant, both data points are consistent with numerous pieces of qualitative feedback from students that these changes were positively received. In particular, one piece of (anonymous) end-of-semester feedback highlighted the beneficial effects of these course policies:

“Prior to [this course], I thought I was good at coding, but this course helped me learn what I didn’t even realize I didn’t know. I’m glad that we had the chance to correct [our assignments], which took some of the pressure off and helped me focus on learning how to code better.”

General Principles for Nurturing Computational Self-Efficacy

In this section, we highlight several broad principles that we have found effective for nurturing computational self-efficacy, and that we believe are broadly applicable for other computational engineering courses.

- Give students opportunities to continue improving upon their code submissions after assignment deadlines, *without releasing official solutions*: This approach, grounded in the principles of mastery-based learning and growth mindset, allows each student to find (and resolve) issues in their code at a pace that is personalized to that student (and without generating anxiety or discouragement upon seeing an “official” solution or solutions that is structured significantly differently from their own). Needless to say, this approach presumes that students have a sufficient level of intrinsic motivation to master the course material, and may be more suitable for graduate-level courses as compared to undergraduate-level courses. This practice may be especially helpful in computational *engineering* courses, where students are responsible for both a significant amount of code development as well as mastery of an underlying body of scientific and engineering principles. By providing students the opportunity to revisit earlier submissions, they can focus more on “in-the-weeds” computational details as their fluency with domain-specific material improves.
- Offer to meet students “where they are” and “as they are,” even while offering them recommendations for improvement: Unlike an introductory course, for which students typically have little to no prior background with the subject matter (and, as a consequence, students are somewhat more likely to pattern their solution methodology after the approaches introduced in class), students in a graduate-level course are more inclined to deploy a (very) broad range of solution approaches. We have found that it is helpful to accommodate as much of this natural diversity as possible, and it is beneficial not to constrain solution approaches at the outset by requiring, e.g., the use of a specific computer language or the use of object-oriented programming. In our experience, most students find

their own way to inquiring about best practices, if given the opportunity to improve upon their previous assignment submissions.

- Provide students "hands-on experience" with computational efficiency (or lack thereof) as early as possible: In the first problem set of this course, students are asked to estimate the computational cost of performing a molecular simulation that explicitly accounts for all of the water molecules in a small (millimeter-scale) droplet of water. Without any optimizations for computational efficiency, this calculation suggests a total simulation time on a laptop that is longer than the age of the Universe. However, in our experience, correctly answering this cautionary tale of a question was insufficient to motivate most students to prioritize the use of efficient programming practices (e.g., the use of a "just-in-time" compiler in Python), or to seek out guidance about how to improve their efficiency. In contradistinction, if a student encounters an estimated runtime of ~100 hours (with perhaps less than 24 hours left before the assignment deadline), they are much more likely to take the need for computational efficiency seriously. Especially in such cases, the availability of assignment resubmission is likely to drive improved learning outcomes, since a substantial restructuring of code may take several days to implement.

Conclusion and Future Directions

In this work, we have discussed several strategies implemented in a graduate-level computational engineering course to help students develop a sense of computational self-efficacy. Although this course has a specific focus on molecular simulation, the general principles described here should be broadly applicable to any course with a significant computational engineering component, and should be fairly easy to implement in graduate-level courses, especially when there is a modest student-to-instructional-staff ratio. Our qualitative and quantitative results indicate that the mastery-based practice of allowing students to iteratively improve and resubmit their code is helpful for improving learning outcomes and for nurturing a sense of computational self-efficacy in students. At its core, our work reinforces the adage, "if you give a person a fish, they will eat for a day; if you teach a person to fish, they will eat for a lifetime": By giving students ample opportunities to "fish" for themselves, they – perhaps unsurprisingly – become more adept at handling bugs.

In the future, it may be intriguing to collect more detailed information on each student's specific computational background (e.g., computational languages and environments known, familiarity with object-oriented programming, number of computational courses previously taken, etc.). Given that most graduate students tend to take a relatively specialized course load (potentially featuring multiple computational engineering courses in a single semester), it would also be interesting to investigate the potential for synergistic (or adversarial) effects between multiple courses taught with different policies regarding assignment resubmission. In the years to come, it will be particularly interesting to study what kinds of educational practices are effective for nurturing computational self-efficacy in students who primarily learned computer programming via online courses during 2020 and 2021.

Appendix: Sample Guidelines for Problem Resubmissions

Below, we provide the text that was provided to students informing them of the opportunity to "take a mulligan" on the third problem set of the course (i.e., to improve and resubmit the problem set).

All problem sets will be returned by the end of this week. You may re-do as much of the "Problem Set the Third!" as you would like to re-do.

Several levels of mulligan are available:

- *"Simple resubmission" (essentially turning in a late and improved version of the assignment): Up to 30% of deducted points will be added back.*
- *"Resubmission + reflection" (in addition to previous material, also attach a memo clearly explaining where you initially erred for each problem, and how you corrected these issues): Up to 60% of deducted points will be added back.*
- *"Resubmission + reflection + hints" (in addition to previous material, also attach a thoughtful set of hints for future students tackling this problem… quality of hints will be judged based on helpfulness and appropriateness): Up to 90% of deducted points will be added back.*

Mulligans are due by 5 PM on Halloween (make sure to take some time and get outside on Halloween evening)!

References

[1] J. K. Burton and S. Magliaro, "Computer Programming and Generalized Problem-Solving Skills:," *Comput. Sch.*, vol. 4, no. 3–4, pp. 63–90, Aug. 1988.

[2] D. W. Dalton and D. A. Goodrum, "The Effects of Computer Programming on Problem-Solving Skills and Attitudes," *J. Educ. Comput. Res.*, vol. 7, no. 4, pp. 483–506, Nov. 1991.

[3] W. Cazzola and D. M. Olivares, "Gradually Learning Programming Supported by a Growable Programming Language," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, 2015, vol. 2, p. 857.

[4] P. E. Anderson, T. Nash, and R. McCauley, "Facilitating Programming Success in Data Science Courses through Gamified Scaffolding and Learn2Mine," in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 2015, pp. 99–104.

[5] P. Karra, "A New Approach to Teaching Programming at Freshman Level in Mechanical Engineering ." ASEE Conferences, Virtual Conference.