Building a Better Computation Course: Achieving Student Success with Active Learning

Nicholas Shane Senske, University of North Carolina at Charlotte

Available at: https://works.bepress.com/nicholas-senske/3/
Introduction

Five years ago, the School of Architecture at UNC Charlotte took the first steps to integrate computation throughout its curriculum\(^1\). Part of this new curriculum is a computational literacy requirement. Computational literacy involves a fundamental set of principles and practices that informs how one should think about and use computing in general and allows one to engage in computing as a way of pursuing design ideas. We believe literacy is a more sustainable notion than focusing on a particular set of tools or software within the curriculum. Computational skills are quickly becoming a requirement in other professional fields, and so every graduate of our school must be able to understand and express themselves computationally, as a means of experiencing and contributing to society\(^2\). Moreover, the fundamentals of computation and how these relate to universal design principles should remain constant - no matter which software or methods are in use - for the foreseeable future. Thus, we believe computational literacy will provide a skillset and mindset that prepares students to design, critique, and innovate computationally well after their formal education is complete.

To help fulfill the requirement of computational literacy in our curriculum, we now teach a foundation course entitled Computational Methods. This paper presents our development process over the past four years and how we adopted new pedagogical methods to improve student affect and learning outcomes.

A False Start

Our initial attempt at teaching computational literacy did not go smoothly. The first iteration of Computational Methods (Fall 2011) failed to meet expectations and was poorly received by many in the class. Students struggled to learn scripting, and although a majority of them completed their homework and projects satisfactorily, reports of confusion were common. In their course evaluations, more than a third of the students questioned the relevance of the course and protested that it should not be required. They could not see any connection between computation and their design education.

This was an unexpected result. The initial framework of Computational Methods appeared to be sound, as it was based upon similar design computing courses from other schools (although
not on this scale) as well as beginning programming texts. Our premise was to help students become computationally literate by teaching the fundamental concepts of computation (e.g. variables, repetition, logic, abstraction, etc.) through exposure to scripting exercises. Students began their week with a lab tutorial to learn and practice basic commands in Processing and, later in the term, Grasshopper (Fig. 1). A lecture followed to expand on these ideas and talk about them in a broader context of computing and design. For homework, the students completed small projects to review the lessons of the week. This structure is common in many design skills courses and the instructor was experienced teaching the material. So what happened?

Traditional vs. Active Learning Methods

On the surface, a tutorial / design sequence (which I will call the “traditional” teaching method moving forward) makes intuitive pedagogical sense. Students need to see a skill performed, then they need to practice it and receive assessment. However, the traditional method does not take into account that computing is not the same kind of skill as drawing or model making. Although digital interfaces require their own type of manual operations, using a computer (even when one is not programming) involves the manipulation of procedural symbol systems. Because of the complex abstract logics involved, these systems do not always follow simple rules of cause and effect. What one sees is not always proof of what has occurred computationally. This can lead to errors and misconceptions. Thus, practicing skills and making artifacts in a traditional manner, as one might drill diagramming or contour drawing skills, may not provide effective feedback for learning.

Another problem with traditional computing instruction is that it tends to focus on software training. This perspective can be counterproductive as it neglects the conceptual foundation that students need to be efficient and effective computational designers. Tutorials can teach software skills, but they do not help students make deeper connections between computation and design ideas. This is an issue because, without a conceptual framework to support them, software skills are often learned in a manner that is narrow and rote: sequences of commands for reliably producing a specific result. As a result, if a student has a problem that does not fit within the commands they are taught, they may be at a loss as to what to do. They may have difficulty generating design ideas outside of the patterns they know. In general, students who learn software this way usually have a poor comprehension of how and why their programs work, as well as a poor problem-solving process with computing. This bodes poorly for them as their skills may not transfer effectively as new tools and methods emerge.

In traditional computing instruction, software tutorials are often followed up with design projects. Ostensibly, these are meant to test computing skills and to help students synthesize and critique what they learned, but this may be inappropriate and damaging when students still only have a superficial grasp of computing. Making is a complex activity and without sufficient practice and mastery of component skills first, the design projects in computing courses tend to engage computation superficially. Worse still, the subjective nature of design and the inability to review student computing processes make it difficult to assess and remediate such problems. The danger of modern computing (but also its strength) is its user-friendliness and lack of transparency. One can make things on a computer with an incomplete or incorrect understanding of the computing involved. Asking students to design computationally too soon, without properly supporting their learning process, often teaches them the wrong lessons.

These criticisms of the tutorial / design method are similar to those of “discovery learning”. This is
a constructivist instructional framework in which students are not provided with answers to questions, but are instead invited to discover ideas, conduct investigations, and teach themselves using their prior knowledge and abilities. In some forms of discovery learning, students are not provided with much initial preparation or guidance. The theory behind discovery learning is that when students discover ideas and solve problems through their own process and effort, they tend to benefit more from the experience. Essentially, this is how many architecture schools expect students to learn computing: learning through making. Instructors provide a brief introduction and the students are supposed to teach themselves in the midst of design projects.

Discovery learning is controversial with many educational researchers today. Some argue that there is no empirical evidence in support of this idea and contest that the theory is incongruent with how learning cognition works. Discovery learning may benefit advanced students, but for novices its lack of structure can lead to unnecessary frustration. The opposing position is that, instead of expecting students to discover learning strategies and concepts on their own, these should be taught to the students in the first place. This is known as direct instruction. A criticism of this method is that it places too much emphasis on the organization of the material and is too dependent upon the instructor.

An option somewhere in the middle of these two pedagogical methods is active learning. The term “active” is used in contrast to “passive” learning experiences. Instead of merely listening or following along in a class, as one does in a typical lab or lecture, the student plays an active role in their own learning process through peer-driven knowledge discovery, problem-solving, and critique (Fig. 2). This may sound like discovery learning or similar to a design studio, but the difference is that an active experience is highly scaffolded and structured (as in direct instruction). Lessons begin by providing students with the specific background knowledge, skills, and strategies they need. This information is carefully organized and presented to help address common misconceptions and reduce unnecessary complexity. Students then deepen their learning through discussing, collaborating, and evaluating ideas together within the context of the new skills and knowledge. Most of this happens during class time, where the instructor can guide, assist, and assess student learning. In the process, active learning not only teaches subject content, it also increases student engagement and helps them become better learners and problem solvers.

The difference between active learning and many other instructional methods is that its effectiveness has been empirically demonstrated across numerous studies. Although we could find no research on the use of active learning in architecture or design computation, it has proven useful for teaching challenging subjects such as physics and engineering. Students in active learning courses tend to have superior final grades and learning retention than students in non-active learning courses. Moreover, they report greater satisfaction with their learning experience. For these reasons, we were interested in studying whether active learning methods could help improve Computational Methods.
Course overview and development

Computational Methods was first offered in the fall of 2011 and has been taught by the same instructor for each of the four years since. It is a required course for third year undergraduates, post-professional graduates in their first year and non-post-professional graduate students in their second year. On average, there are 50-70 students in the course, with a proportion of 60% undergraduates to 40% graduate students. For the purpose of the research described in this paper, it is important to establish that the overall framework of the course has stayed intact for the past four years. The goals and topics of the course have not changed substantially, only the instructional methods. The course schedule and contact hours have remained consistent for four years, as well. The course meets twice a week – once for a 25 person lab seminar and again for a lecture with the entire class -- for about three hours total.

The first iteration of Computational Methods (2011) was described earlier in this paper as a traditional computing pedagogy. The instructors used lab sessions to present skills tutorials, gave weekly design assignments, and taught concepts in a lecture format. When this version was unsuccessful, it was clear that the course needed to be redeveloped. Based upon their feedback, the students were most dissatisfied that they were required to learn scripting. Many found the material too difficult to follow. This is common for most novices starting out with any form of programming. Indeed, it has been well established in other research that programming is a challenging skill to learn – even for many new computer science students. However, scripting is essential to thinking about and practicing computation, and so it had to stay part of the class. We needed to rethink how to teach it.

Our response for the second iteration of the course (Fall 2012 and 2013) was to develop what we now call “active labs.” Before each lab, students prepare by watching skills videos online. These replace the content of the previous lab tutorials, but have additional benefits. In particular, they allow students to control the pace of their own learning and conveniently review complex material. In the active lab, students continue to learn at their own pace, as they work together in pairs to solve problem sets using their scripting skills. The lab becomes a more social environment, which helps lessen students’ apprehension about programming. It also allows for more interaction with the instructor, who can quickly address technical problems and other questions. Labs guide students through the development of a personal computational design process by modeling methods of investigation, iteration, testing, and refinement. They also challenge students to explain how and why their programs function. Essentially, the goal of the in-lab assignments is to teach what typical design assignments often leave out.

There are some design projects in Computational Methods, but these are also highly scaffolded. For example, one project asks students to reverse-engineer a parametric gesture from a set of precedent buildings, explain the procedure graphically, and then produce new variations of the gesture with their script. This is an effective project because it does not ask students to come up with an original idea (which would be challenging given how little they know at this point) and places emphasis on clear process and explanation. The current final project has students test and refine a building element using some form of metric-based analysis (e.g., light, heat, stress, etc.). This is a practical and rigorous use of scripting that avoids the production of parametric aesthetics for its own sake. Overall, the design projects we assign are open ended enough to provide students with some creative freedom, but organized so that students do not struggle because they do not have a clear sense of what to do.
While the feedback from the second iteration of the course was mostly positive, there was one exception. Students who were used to the active lab sessions did not find the lectures engaging. After interacting together in the lab, they could not easily switch back to passive learning. Therefore, in our third iteration of Computational Methods (Fall 2014), we attempted to address this though the creation of an “active lecture.” Similar to the lab format, students come prepared for lecture by watching materials online. These videos replace much of the lecture content covered in previous versions of the course. Instead of watching the instructor deliver the lecture in person, the recovered class time is used for discussions, follow-up on difficult problems or concepts, and for brief activities designed to reflect upon and deepen students’ understanding of the material. In our active lectures, students not only have the opportunity to review and remediate what they learned, the new format also asks them to think about it with multiple perspectives. It is a far richer experience this way. Instead of a classroom where students expect to be given content knowledge, we turned the lecture into a social environment where students create and test their own knowledge. This is the very definition of active learning.

Fig 3. Students using active learning classroom technology (Author’s photo)

Our effort to redesign lectures was assisted by the completion of a new learning space on campus with technology to support active learning (Fig 3.). In addition to numerous projection screens and whiteboard surfaces throughout the space, the classroom features small group tables, each with embedded laptops, microphones, and a large monitor to support collaboration. The tables allowed us to have discussions in small groups that could be shared with the whole class through the microphones. Groups also used their laptops to create short presentations on concepts and precedent examples. A limitation of the space is that it would not support our scripting software (Grasshopper). This may have turned out to be a positive event, because it encouraged us to come up with other ways of thinking about and discussing scripts: for example, making lists of ideas on whiteboards and diagramming script logic using a touchscreen paint program. This gave the students a different way of thinking about computation beyond the software, which may have helped them understand it better. Overall, the variety of activities and high level of student interactions kept the class engaged. Students were constantly making, talking, and moving around the room. The technology in the classroom provided us with more options but was not the most important factor. The difference came from changing the lecture to involve the whole class.

In four years, Computational Methods progressed from a traditional computing class to one that uses active learning throughout. This process was driven by student feedback and data collected as part of an ongoing study of the course’s effectiveness. In the remaining sections, I describe our methods and findings from this research.

Research Methodology

Since Computational Methods was introduced as part of our new curriculum, we have been conducting a long-term study to assess its
effectiveness. Throughout this time, we maintained a consistent set of data-collecting instruments which have helped us track student performance and affect in response to pedagogical interventions over the past four years. The first of these instruments is the institute-standard student evaluations. These are electronic surveys with multiple-choice questions and a brief written section for student comments. We also collect scores from submitted assignments and final grades. The second set of instruments is a pair of voluntary online surveys created to measure student attitudes and feedback, pre- and post-class. These surveys use a combination of multiple choice, Likert-style (Agree/Disagree, etc.), and short answer questions. For the purposes of comparing the three versions of the course, we present the data from the second iteration (2012 and 2013) classes as an average.

Results and Discussion

The introduction of active learning labs in the second iteration of Computational Methods made a significant improvement in student outcomes compared to the first iteration of the course. To be certain that the pedagogical interventions were the cause, we taught the same course a second time and were pleased to find that the data remained consistent. The two most important findings from our comparison of the first and second iteration data were that the average student score and pass rate increased (Fig. 4) and that students overwhelmingly found the material more relevant and enjoyable than before (Fig 5). These positive results gave us some assurance that Computational Methods could succeed as a required course and indicated that active labs made a difference in student outcomes.

Adding active lectures in the third iteration of the course seemed to improve upon these outcomes further (Fig. 4). While the average student score remained steady compared to previous versions, the passing percentage rose to 100%. In addition, the scores in our affect surveys improved an average of 2.4% (Fig. 5). Because this is the first time teaching the active lectures, these results are inconclusive at the moment, but the fact that we were able to maintain our high scores from the previous iteration and improve on them even slightly is a positive indication that active lectures are a useful addition to the course.

Overall, the majority of students in the third iteration (~80%) told us that they found the active learning activities both helpful and enjoyable (Fig. 6). The skills videos were considered the most helpful resource, followed by the active labs. The active lectures seemed to be the least helpful, perhaps because they were still experimental. However, they still received high marks. 80% of
students surveyed said they preferred the active lectures to traditional lectures. When asked whether the active lectures should remain part of computational methods, 93% of students answered in the affirmative. This gives us confidence that the latest changes were well received by our students.

<table>
<thead>
<tr>
<th>Resources survey - 3rd iteration</th>
<th>Helpful?</th>
<th>Enjoyable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skills videos</td>
<td>62.0%</td>
<td>84.8%</td>
</tr>
<tr>
<td>Lecture videos</td>
<td>84.8%</td>
<td>80.8%</td>
</tr>
<tr>
<td>Active labs</td>
<td>87.0%</td>
<td>85.1%</td>
</tr>
<tr>
<td>Active lectures</td>
<td>80.4%</td>
<td>80.4%</td>
</tr>
</tbody>
</table>

Fig 6. Learning resources survey, % of students answering Strong Agree / Agree to statements

Why did the changes to the course lead to improvements? This is difficult to say for certain. We believe it is because of the shift to learner-centered methods which are more scaffolded, socially enacted, and empowering. A deeper study of computational principles coupled with opportunities for discussion and reflection seem to have transformed student attitudes about computation and its relevance to architecture. For example, in their written evaluations, one-third of students in their first iteration specifically mentioned software (e.g. Grasshopper / Revit) in their comments; only four students made any positive statements about the relevance of Computational Methods. In the second iteration, 20% of students mentioned software and a third cited the relevance of the topic. By the third iteration, only one student specifically mentioned a software program; half of the class explicitly said they thought the course was relevant. The students used phrases like [computation is a] “new way of thinking” and a “way of thinking about design.” This trend appears promising, as it suggests that the introduction of active learning helped us make the argument that computation is more than software and not a specialist practice.

Summary

The data collected from Computational Methods showed improvements in student outcomes and engagement after pedagogical interventions using active learning methods. While the improvements in the data could be explained by the instructor’s increasing experience, the differences before and after the interventions are substantial enough that they cannot be explained by this alone. Because our results appear to be consistent with those of other active learning studies, this leads us to conclude that active learning is responsible for the positive changes in student response to Computational Methods. Judging by our students’ work and qualitative evaluations, these new methods have helped us make a difficult subject accessible and interesting to diverse cohorts of graduate and undergraduate students. In general, our students learn, appreciate, and like what they study in Computational Methods.

Future Work

A challenge in our current research is that we cannot yet empirically study how well students understand and can apply computation to their design work. Our assignments provide a general measurement of students’ subject mastery, but we do not know how well they retain their skills and where gaps in their knowledge persist. Unfortunately, measuring students’ computational thinking abilities is a challenge. Standardized tests of computing ability are not useful when measuring the effects of computational thinking upon design activities with unique solutions, and examining designed artifacts does not reveal enough about the thinking used to create them. Until we find a superior method, it remains difficult to effectively test for computational literacy. To address this problem, we are working with faculty from education and cognitive science to develop experiments that should lead to more effective means of assessment.
An additional challenge we still face is how to pursue computational literacy in later studios and courses. While a few students attempt sophisticated computational approaches in later projects, the majority either choose not to or fail to see the opportunities. A possible reason for this is that the course does not come early enough in the curriculum, particularly for undergraduates. By the end of Computational Methods, students are still novice learners in the subject. They need more time and opportunities to develop what they learned. Another reason is that we have not yet sufficiently integrated computational ideas in later courses. The onus is often on the student to apply computation, because the instructors or their prompts do not call for it explicitly. Introducing more scaffolded projects that encourage or require computation would help students practice and refine their abilities. Computational literacy is a long-term curricular effort that we are working to coordinate better. The takeaway from our experience is that a single course is not enough.

Conclusion

We hope our experience integrating computational literacy into the UNCC curriculum may provide a useful example for others. Although our students responded well to active learning, we do not claim it is the best or only way to teach computation for all types of student. A well-planned development process is equally as important as sound pedagogical theory. The key to this process is abandoning unfounded theories and making instructional choices based upon empirical evidence. If computational literacy for all design students is a long-term goal, more rigorous pedagogical research by the discipline is the only way forward.

Notes:


