Work in progress: A first year common course on computational problem solving and programming

Dr. Bruce W Char, Drexel University (Computing)

Bruce Char is Professor of Computer Science in the Department of Computing of the College of Computing and Informatics at Drexel University.

Thomas Hewett is Professor of Psychology (emeritus) and Computer Science at Drexel University.

Dr. Thomas T. Hewett, Drexel University

Tom Hewett is Professor Emeritus of Psychology and of Computer Science at Drexel University. His teaching included courses on Cognitive Psychology, Problem Solving and Creativity, the Psychology of Human-Computer Interaction, and the Psychology of Interaction Design. In addition, he has taught one-day professional development courses at both national and international conferences, and has participated in post-academic training for software engineers. Tom has worked on the design and development of several software projects and several pieces of commercial courseware. Some research papers have focused on the evaluation of interactive computing systems and the impact of evaluation on design. Other research papers have explored some of the pedagogical and institutional implications of universal student access to personal computers. In addition, he has given invited plenary addresses at international conferences. Tom chaired the ACM SIGCHI Curriculum Development Group which proposed the first nationally recognized curriculum for the study of Human-Computer Interaction. Tom's conference organizing work includes being Co-Chair of the CHI '94 Conference on Human Factors in Computing Systems and Program Chair for the 2013 Creativiey and Cognition Conference.

A first year common course on computational problem solving and programming

Abstract

This is a report on work-in-progress for an entry-level course, Engineering Computation Lab, in which engineering and other STEM students learn about computational problem-solving and programming. It provides a hybrid (on-line and in-person) environment for learning introductory programming and computational problem-solving. It runs at scale, serving 800-1000 engineering students per term. Pedagogically, it uses active and problem-based learning using contexts more oriented towards the needs of engineering students than typical generic "intro programming" courses. Autograded exercises and on-line access to information have been key to feasible operation at scale. Learning how to operate effectively and smoothly at scale across a variety of lead instructor preferences, with the particular needs for computation by engineering students has been the continuing challenge. We report on our experiences, lessons learned, and plans for the future as we revise the course.

Course objectives

Use of computation is indisputably part of every engineer's foundational training. However, there does not appear to be a consensus on the extent of such training, or its outcomes. Training for professional software developers (as evidenced by what it would take to be seriously considered for a professional software development position nowadays) would seem to include the equivalent of at least several terms of courses to achieve a working knowledge of software development: programming in two or more languages, data structures, performance analysis, software design, and basic principles of software engineering such as testing, documentation, and archival storage. However, the conventional undergraduate engineering degree is already full of other mandated science and discipline-specific course work. Until the day arrives where enough time is given to establish mastery of software development, course designers would seem to need to settle for the goal of introduction: initial experience with of the skills and knowledge needed to create and use software to solve problems typical of an engineer's work. This includes:

- Concepts of simple computation: command-based access to technical functionality; scripting and introductory programming (variables, control statements, functions).
- Application of computation to engineering: concepts of simple mathematical modeling and simulation, and their use to solve engineering design problems.
- Software engineering: how to build software, get it working quickly, and having confidence that it works well. Also, how to better amortize the cost of building software by designing for reuse.

Mastery of these concepts is clearly beyond a single course, or even a year long sequence of courses. Yet postponement to sophomore or junior year blocks access to even simple computation skills and concepts in the first years, which blocks more sophisticated use of software when it might be used by some for educational benefit.

Engineering Computation Lab, 2006-2013

The course was originated and underwent pilot development in 2005 under Jeremy Johnson with an enrollment consisting primarily of first-year Computer Science majors. The first full deployment of the course began in 2006 at a scale of approximately 600 students/term. In 2008 we made the transition into a course that was a hybrid of in-person lab activities and out-of-class on-line autograded exercises with approximately 800 students per term.. The course operates during the three quarters of our institution's academic year as a series of 15 two-hour class meetings (one unit credit per quarter) to better achieve the benefits of spaced learning[1]. During a term there were 4 two hour labs, with the 5th meeting a two hour proficiency exam. There were an on-line pre-lab prep quiz, and a post-lab on-line homework assignment. The course typically ran as approximately 30 lab sections of 30-35 students, across 20 different time periods. This is an example of a "flipped classroom," in that most of the contact time was spent in active learning from lab activities.

Choice of language

The first version of the course used Maple[2] as the computation system and programming language. Maple was selected for several reasons.

1. Maple is interactive, similar to systems such as Python, MATLAB or Mathematica that allow immediate execution and display of a single operation without a compilation phase. This leads to more immediate feedback and interaction, more suited to developing a "personal computing" – getting results that are of interest and immediately useful for an individual's work. Pedagogically, it allows students to get at least simple results immediately, with incremental growth from that point.
2. Maple has a large library of STEM procedures, permitting use of sophisticated technical computing without extensive user programming. Typical small-scale software development consists of writing the scripting connecting invocations of library procedures, and providing the user interface programming that allows facile comprehension of computed results through tabular listing, plots and animations, etc.
3. Maple's standard user interface are interactive worksheets allow text, 2-D mathematics, code, and graphics to be combined in a WYSIWYG fashion. A publishing mechanism allows generation of html or pdf from worksheets. This allowed us to write the lab and course notes[3] using the same tools and in the same environment that the students used for their own work. For example, lab work could be entered by students within the same document that distributed the directions. The availability of "clickable math"[4] in the interface allowed an incremental approach to the introduction of linear expression and command syntax used in most conventional programming languages, providing a gentler learning curve to the syntax that can often be a work bottleneck for beginners.
4. The course's on-line autograding system, Maple T.A™ uses the same programming language. Since Maple T.A. allowed questions to accept student responses using Maple syntax, this made it particularly easy to ask autograded programming questions that required students to submit code as their answer.

Lab activities

Before each lab, students were expected to read course notes[3], lab notes[5] and take an on-line pre-lab prep quiz, to ensure familiarity with the programming concepts needed and activities that they would be asked to do in the lab. Labs typically started with a short (20 minute) lab overview presented by the lab instructor. This was their opportunity to emphasize what the students should have begun to understand through the readings by taking the pre-lab quiz. Labs typically led students through a number of tasks. Work was in groups of two or three students, based on evidence-based research that such work enables better learning compared to individual work[6,7].

Often there would be "scaffold" programming for the students to complete. Typical labs would introduce some computational or programming concepts and ask the students to complete the programming for the following kinds of tasks:

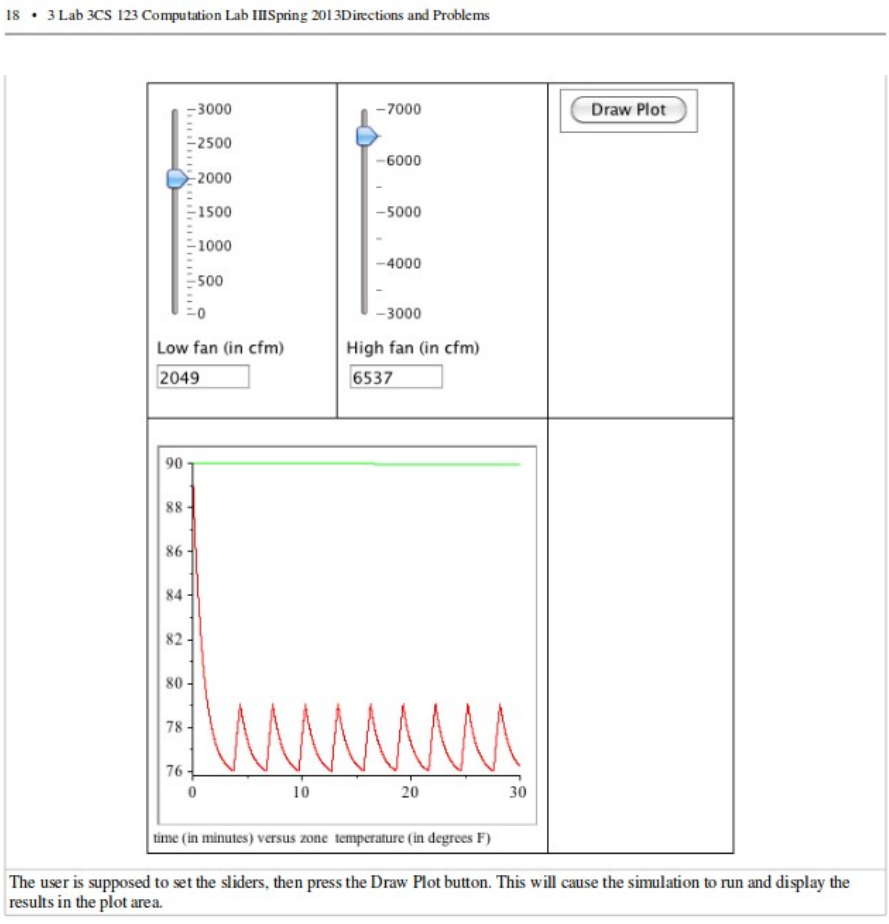

Illustration 1: Lab exercise from [5]. In the previous lab, students had completed scaffolding of an HVAC time-step simulation. In this lab, they learn how to steer simulation runs and display graphical output by building a GUI in Maple that invokes the simulation.

Question:

Defining Custom Functions

This question references Section 7.7 of the textbook starting on page 100.

By filling in the blanks below, enter the arrow (->) textual version of a Maple function definition for the following *mathematical function below*.

$$H(t) = 23.51000000 - 23.51000000\, e^{-1.800000000\, t}.$$

Your job here is to translate between "mathematical notation" and "Maple notation" for function definition.

Define the function using the arrow (->) method.

[ ] := [ ]

Illustration 2: Sample coding problem from Maple T.A.

1. Create a plot for the population growth of a species in an ecosystem, and make predictions based on the graphical and numerical results.
2. Create an animation of the trajectory of a human cannon ball (from [8], pp. 462-465), based on given formulae for velocity/position. Variations included an improved model taking into account wind resistance, or generating a position plot of a bouncing ball.
3. Calculate a least-squares trendline from given time-temperature, or estimated-actual distance measured from a sensor, and to answer situational questions.
4. Write a control program for an autonomous car simulator, and test it on varying terrain with a common feature.
5. Calculate and plot the dynamic behavior of a chemical reaction involving four chemicals described through difference equations (approximating differential equations).
6. Calculate the area under a curve by developing a piecewise polynomial spline fit to data by using symbolic definite integration (originated by [9]). Use this to calculate the power developed by time-velocity measurements of a baseball batter's swing.
7. Complete and extend a time-step simulation of an HVAC system (see Illustration 1). Analyze a design space by varying heat/cooling parameters. Design a control program for a ventilation fan and observe homeostatic temperature behavior based on the fan control parameters.

The lab work included getting the student to consider and answer questions that involved interpreting and steering the use of programs they had developed, rather than just getting coding to pass specified tests. This is one of the major differences of this course from a generic "intro programming" course. It is ability to design, implement, and then use computation to solve problems that is most important to STEM students who need to develop programming skills. We found that doing this does not occur "for free" – there are time tradeoffs involved in having students learn about using computation for problem-solving, rather than learning additional

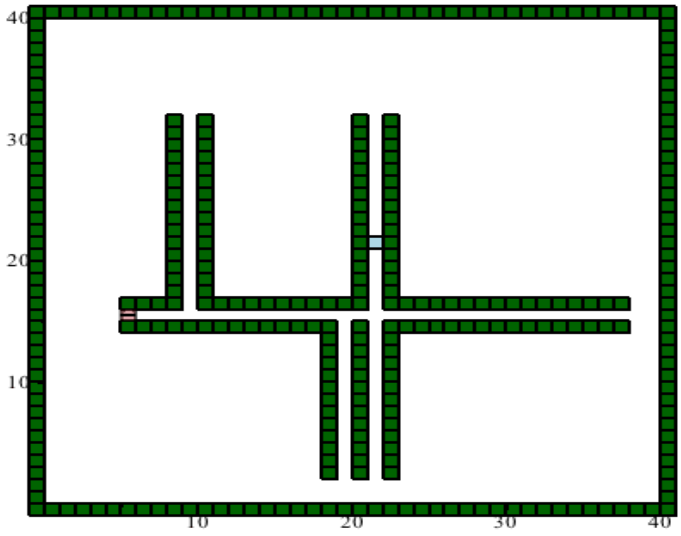

Illustration 3: Autonomous Car Simulator Scenario. Students were given an API for the simulator and asked to write a Maple control program to navigate through a family of scenarios. The white space is the "road" while the green are road shoulders. The blue square is the target destination.

programming features.

While this first course does not provide a complete education in computational engineering, we believe it provided a way to get students engaged early with the application of computation skills. This makes possible follow-on courses where the integration of programming and problem-solving skills can be mastered through repeated, increasingly complex cycles of instruction.

Autograded problems: pre-lab and post-lab activities

Maple T.A.[10] is a proprietary on-line system for administering on-line exercises and tests. Like other on-line systems, it allows instructor-constructed questions of the conventional sort – multiple choice, fill-in-the blank, matching, etc. Studies have indicated the effectiveness of on-line training for programming [11]. We also took advantage of the cost/time saving features of centralized computerized grading in reduced staff resources for grading and its administration. The reason why we chose it over more common systems is that the Maple engine can be called on for problem generation and answer checking, giving it particular strength for work the STEM area.

Maple T.A. was used in the course as a vehicle for creating and delivering questions for pre- and post-lab activities, as well as the in-class proficiency exam at the end of each term. After a few years' development, we came up with a battery of prospective problems that we could offer on a rotating basis, requiring only incremental or evolutionary revision from year to year. We relied on the instructor- programmability to develop questions appropriate for the problem-solving/ programming dual nature of the work, at scale. In particular, we developed techniques for delivering problem variants to students using random selection, random parameter values, and

random variation of the problem and its solution algorithm as described further in[12,13] Having found a problem area, we would develop several questions that could be answered through computation from a model or code. We would write the program that could check solutions for a variant. When the student asked for their assignment, the system would generate the particular version of the problem, along with the problem description and answer-checking specific to that version.

Illustration 3 shows a scenario for an autonomous car controller problem. Students in lab became familiar with an API for a simulated car controller, which allowed a car to move and turn based on "stylized" sensor readings of its proximate environment. In the autograded problems, they built upon their experience to write control programs for obstacles more ambitious than those they tackled in lab. The variant creation would change the obstacle course so that different sequences of actions and turns were necessary. The presentation would include a dynamic generation of the textual description of the scenario, as well as animations illustrating the obstacle courses that provided the tests for the student controller.

Because of the bi-weekly nature of the lab meetings, there was a week where students did not meet staff in lab. This week was used for makeup labs and for completion of autograded homework exercises. The nature of these exercises combined simple questions about knowledge of language syntax, or one-line code solutions, with problems that required students to extend lab programming to handle new situations, or to create small programs or scripts that were expected to be modifications or augmentations of programming given in the lab or in the course notes. Examples might include:

- Calculating the voltage across a resistor with a given amount of current flowing through it.
- Asking what angle and velocity would work to make the cannonball travel through a ring suspended a certain distance and height.
- Answering questions that predict time or quantity from a trendline formula constructed from given data.
- Figuring out how to compose given plotting functions to create a specified picture, and entering the function invocation.
- Creating control programs to handle additional scenarios for the autonomous car.

About 120 problems were created for the course. Many of these problems requiring the student to take multiple steps, intended as a kind of problem-based learning [14].

Designing autograded questions

Instructor programmability of the autograder, as Maple T.A. allows, opens the door to more sophisticated question design, as described in[13]. We have found the following to be important considerations in our design activities:

1. *What kind of question do you want to ask?* Given the particular selection of topics for the week, there were numerous possibilities: knowledge acquisition/review from readings (where the humble true/false question was often good enough), problem-solving using

problems similar to ones covered in lab or the readings, exercises that would require result interpretation or reflective thinking, problem-solving that would require adaptation and transference of learning, etc.

2. *How much time should students expect to the week's autograded work will take, and how will you make your question selection fit within that time budget?* Despite its use of autograding, our course emphasizes learning through fixed amounts of lab time in social interaction with staff and lab partners. There was not the development budget nor the inclination to use autograding as a kind of "intelligent personal tutor[15,16]" whereby a student works many hours being guided through programmed instruction until mastery of a skill is detected. Nevertheless, it was easy to come up with questions that would require far more time than the students thought they had for the course. In conventional instruction limiting the assigned work is also a way to avoid overloading the amount of grading effort for the staff, but with autograding this is not the case. The "retry until success" work ethic also may require more time than conventional homework. In a course like this that combines both skill-building practice, software development, and problem-based learning, it is important to be aware that problems may, by design, vary greatly in the length of time to do them.
3. *What application area should the question be about?* The topics covered in a particular lab cycle would often be fertile grounds for any number of questions. For example, in the "human cannonball" portion of the course, one could come up with a number of variants of the basic problem, requiring different formulas, different boundary or starting conditions, different kinds of information. To keep current the idea that the knowledge and skill-building in the course was intended to have transferability to situations not literally the same as covered in lab, problems on other topics would be used. The programming of the autograded problem would be used to present more information about the new domain, rather than relying on student familiarity with the problem area, sometimes including graphs, formulae, or animations.
4. *How much scaffolding do you want to put into the question?* Some students had a lot of difficulty with problems requiring multiple steps to solve them, because they could not figure out how to get to the goal from the start. The hinting mechanism of Maple T.A. was better at giving short, one sentence general advice. It was also difficult to sense what a student's conceptual problems were from the numeric or programming answers they submitted. As we developed experience with certain problems, we were able to develop on-line notes that were helpful to many (although at the cost of not providing as full an experience at self-directed discovery of solutions). We found useful to break up longer problems into multiple parts, asking for intermediate results or realizations before to the final answers. With this approach, students could see that they were making progress even if not getting to the end. Visits to the walk-in clinics became more productive because the staff had more information about where a student was running into problems. Generating a multi-step problem just means modifying the solution algorithm for the final result to output intermediate results it computes along the way. The sequence of results can then be used in a multi-part question.
5. *What information does this programming of a problem solution need, and where will the students find it?* In the later portions of the course, some of the problems relied on the software packages developed for the course, such as the autonomous car simulator. To avoid unintended or misinformed use of alternatives (e.g. older versions of the packages

that were somehow still on someone's web pages), the directions needed to be specific about providing hyperlinks to the intended files.

6. *How much effort will it be to test the questions before release?* Testing questions for comprehensibility and correct operation is an important fact of life when operating at scale. Even if only 5% of the students run into problems or ask questions in our course, that will generate 40-50 situations that require a staff response. We had to treat our autograded questions like software engineering, allocating testers and time for testing before release.
7. *How much computer time does it take to generate the question, and to check the solution?* In this era of gigahertz/gigabyte computers, it is easy to think that any problem suitable for introductory students would use negligible amounts of computing time. However, larger problems (due to a realistic application, or for anti-gaming purposes, see next section) requiring dynamic generation to produce can take a significant fraction of a second to create on-the-fly. An example of this would be a problem generating a sophisticated animation to help present an explanation of a new application area. While a half second of computer time is no issue for autograding a class of 30 students, when operating at scale it can lead to server saturation and unacceptable response times. One can avoid problem generation bottlenecks by looking for alternatives that require less computation, or by precomputing hundreds or thousands of variants and storing them on the server. Then most of generation time is spent in modest amounts of disk access, e.g. retrieving an animation file rather than computing many frames of a complex scene.

Autograding and "gaming the system"

By "gaming", we mean the activities of students who operate the autograding system but wish to bypass learning or trying to solve a problem with an authentic general-purpose approach[17]. An example of this might be a student who, after learning that a testing algorithm checks a student program on three fixed inputs, just writes a series of if statements that deliver the correct statement in only those three cases, foregoing the programming that might provide the correct answer in any other case. The use of variants, and requiring solutions to several different versions of the problem, is itself a way to encourage students to work out the solution on their own rather than relying on look up or trying to get by using verbatim responses they get from classmates or web site look up.

While allowing retries is a good way of encouraging students to pursue correct answers if success is not immediate, it also can be gamed using exhaustive trial and error. Unfortunately in the initial years of the course Maple T.A. did not have a convenient way of setting up an assignment so that some parts would allow unlimited retries and other sections (such as with questions that are true/false or multiple choice) that do not. True/false or multiple choice questions are obvious kinds of questions that could be gamed with unlimited retries. Other examples: a math-oriented problem that asked for a fill-in-the-blank integer solution where it was clear that the only sensible values would be between 1 and 10; an optimization problem requiring a three digit answer that could be gamed by doing a plot and then exhaustively trying all numbers around where the plot indicates the optimum occurs. Sometimes in trying to simplify the work of students by presenting a situation simple enough for a student to quickly understand, the situation be solved through short-cuts that avoid the intended techniques being intended.

To avoid asking questions that can be easily solved through exhaustive trial-and-error, create problems where a large amount of manual effort is needed for likely success. For example, changing the precision required or range of plausibly correct answers so that there are a minimum of hundreds of possibilities to try.

Another kind of "gaming" are students who will try (authentically) to answer an autograded question without using the techniques of the course. An example for this course is the desire of some students to stick with finding answers with hand calculators (which they are experienced in and confident about using from high school work) rather than learning programming. For this reason we typically designed problems to a) require working with data sets that were too large to solve through eyeballing or to key into a hand calculator (i.e. don't ask the student to write a program to add together numbers and then give them only three number to process), b) ask for solution to two or three versions of the problem that differed only in minor ways parametrically, to favor use of techniques such as programming where procedural reuse is easy, and c) were situations where use of computation would be authentically better than other approaches such as logic, commonsense reasoning, visual inspection, or lookup. The vulnerability again arises from the fact that autograders can't see the work the students do to get their answer, combined with an instructor's natural inclination to give students a simple situation that does not make them work too long to understand what is being asked.

Proficiency exam

Each of the three quarters of the original version of the course ended with a proctored proficiency exam in the lab section. Use of computation tools and their on-line help, along with information on the course web site during the exam was expected, but proctoring included electronic and personal measures (network blocks, desktop monitoring, walk-by inspection) to discourage unauthorized access to information.

The exam asked students to demonstrate their programming and problem-solving skills by doing so during the test. The exam given to each section consisted of a subset of the pre- and post- lab autograded exercises. All exercises for the term were made available for practice ahead of exam wee, but students did not know which questions would be used for their section's exam until they took the exam. Thus, all sections across all time periods were on equal footing as far as knowing what would be asked, and how to prepare for them. It also made it possible to give makeup exams for excused absences or exceptional circumstances. During the exam, students were allowed access to the electronic course and lab notes stored on the course web site, as well as on-line help built into the version of Maple running on their computer. They were expected to construct Maple scripts on their computer to calculate the results the Maple T.A. problems requested. submit them to Maple T.A. Since most exercises came in numerous variants, the exam tested whether student had become familiar with the programming and technical concepts to be able to recreate and solve a problem that they had supposedly mastered. Because the exam was autograded, results were available to students as they exited the exam.

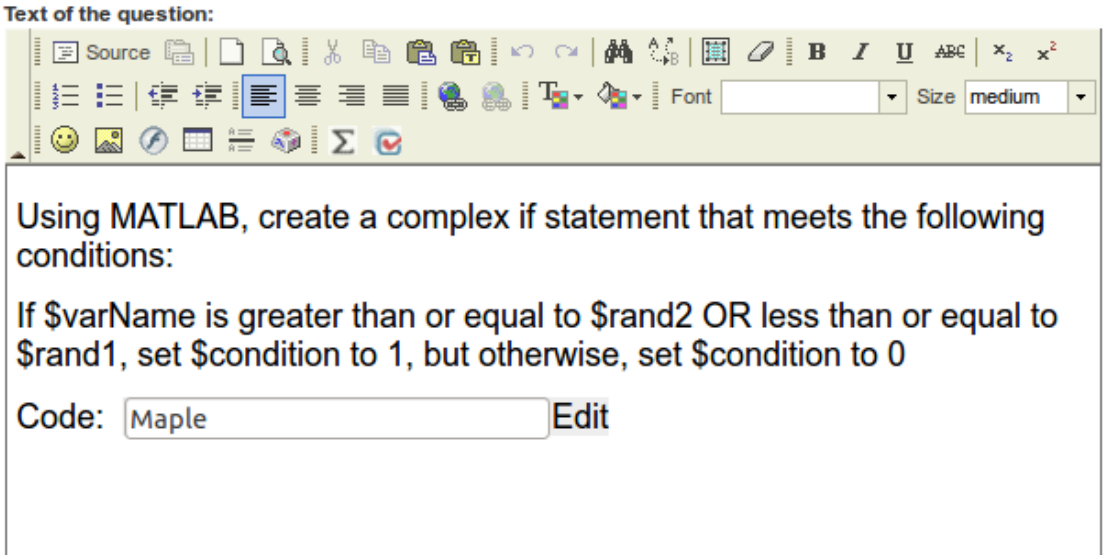

Illustration 4: Example Maple T.A. question for MATLAB-based version of the course. The student response is translated from MATLAB syntax to Maple syntax automatically so that the Maple engine within Maple T.A. can check the symbolic answer.

Having a proctored exam worth a significant component of the final grade was also an anti-gaming measure, in that it detected the absence of genuine learning during the practice afforded by the lab and autograded exercises.

Lessons learned from the first version of the course – strengths

Much of the effort in the past few years went into the authoring the founding materials. For example, even though extensive use of autograding made it possible to run the course with the staffing resources given, instructor authorship of autograded problems that come in variants is an exercise in software engineering: each problem took approximately five hours to develop and test to the point of release. While this amount of effort per problem would be prohibitively expensive for a "use once" problem, the scale and reusability of the problems across multiple exercises and tests, and across multiple course offerings, provided a solid economic rationale for this work. In "steady state", one would expect question development to be more incremental and feasible within the time budget conventionally allocated for development for large "flagship" courses. Development of the lab and course notes required several revisions to improve clarity of exposition, and achieve appropriate pacing of the work. Topics were reordered to try to satisfy external needs, such as the need for earlier control statements and user-developed procedures arising from a LEGO robot programming project being handled in a concurrent Engineering Design course.

Because we viewed the course as constantly evolving over its first few years as we grew in experience and understanding, we sought information expediently for formative rather than summative evaluation.

We found that the package of activities (labs, reliance on written materials and active learning) were effective at allowing to acquire some of the skills we were providing training for in the course. For example, we introduced the Least Squares data fitting concept in a lab problem tackled by students in groups, with the same grade given collectively to each group member. Subsequent autograded results for individuals found that the skill seemed to be acquired by over 80% of students, and with persistent results when the question was asked again a term later.

Event	Test average
Fall, 2010 post-lab quiz (874 students)	87.0%
Fall, 2010, end of term proficiency exam (551 students)	81.2%
Winter, 2010 review quiz (802 students)	86.3%

Table 1: Performance on a least squares data fitting problem (Not all students were given the problem on the proficiency exam to avoid predictability across exam time periods.)

the students to pass the proficiency exams averaging 80% as indicated in Table 2. The decline in the scores from progressive terms, which we attribute to the increasing difficulty of the work, similar to that reported in other introductory programming courses[18]. However, the term-by-term achievement scores remained steady over several years' operation of the course.

CS121 (first term)	CS 122 (second term)	CS 123 (third term)
89%	80%	76%

Table 2: Proficiency exam scores for 2011-2012 year, approx. 800 students

Our system logs for Maple T.A. indicated that the most popular times for work were evenings. By allowing retries, we believed we encourage a "attempt until success" work standard that is crucial to success in programming. The lack of sophisticated feedback by the autograder was frustrating at times to some students, since they did not have any easy way to proceed after submitting a solution and having it marked wrong. However, the use of autograding allowed us to reallocate staff resources from grading to providing walk-in clinic hours to serving students who did have difficulties.

A typical year's operation saw over 122,000 problems graded automatically – not including the additional grading resulting from student retries. We attempted to keep the entire class on a single schedule of due dates, but this imposed significant swings in the load on the autograding system. Fortunately our system administrators were able to deploy adequate server power to handle our size class. Nevertheless, system performance requires careful attention in courses where significant resources are needed for autograding.

Lessons learned from the first version of the course – limitations of the original format

Maple T.A.'s grading of submitted programming had significant limitations. Input is inconvenient (cut and paste of the program into a small text box). If a program would fail, the autograder result would not include any of the standard feedback and error messages that the desktop version of Maple would give for the same program. It did not safely sandbox certain kinds of "runaway" computations which would cause the server to become unresponsive. Because of these issues, we tended to ask "applied" problems where the input to Maple T.A. consisted of results computed by students after developing and executing code on their own

computers. This meant that the course was weak in the area of drilling students on mastery of development of small programs, away from lab. While many students (as indicated by the course marks) were able to learn programming despite the lack of good low-level automatic feedback, we saw that there were some students who were not well-served by this deficiency.

The limited time available for course work, in class and outside, meant that even with autograding there was not enough time to exercise all skills separately and then in integration. This produces a less-than-ideal situation for a subject where there are lengthy chains of interdependent skills. In some cases, we were forced to fall back on the hope that what we required would be sufficient for some students, and by providing social reinforcement many of the rest to go beyond required work to the level needed to complete their mastery.

Because of the difficulties of executing student program submissions, we did not pursue the important notion that a program can be evaluated in several different ways – correctness, run-time efficiency, quality of coding style, quality of design, and how well the computed answers satisfy the need of the user. We think that the effort/feedback experience of trying to satisfy an autograder on multiple criteria simultaneously would be worthwhile for students, but it remains as yet unimplemented in our course.

Our impressions anecdotally from the institutionally mandated course surveys and from discussions with individuals is that some students struggle with the need for original synthetic thought that is the basis of being successful programming; they seem to be more comfortable with STEM work that is recall- or pattern-matching based. They also find the abstraction of both mathematical modeling and the dynamic state changes of a computer program to be hard to work with, with a resulting decline in motivation. We realized that finding ways of increasing motivation and finding ways of reducing the stress of the cognitive load of abstraction might be a way to having more students succeed in the learning tasks.

Finally, the current limitations of autograding technology meant that the assigned work was "directed" – doing particular things for particular results. While much of introductory college-level course work is of that nature,, it meant that questions where the answers could not be anticipated or evaluated easily through an algorithm. Open-ended problems or project-based work[19–21] could not be accommodated. While this is no worse than courses that use problems from the back of textbook, nevertheless, the appeal of project-based work is that some students respond better to circumstances where authentic original work is necessary.

Engineering Computation Lab 2.0: revision and refinement.

The course is now in the process of revision, keeping its strengths while addressing limitations and weaknesses.

1. There are more contact hours. The class is now run two terms instead of three, but with weekly instead of bi-weekly labs and a weekly one hour lecture. This increases the number of contact hours from 30 to 60, of which at least 40 are active learning. This should increase the inventory of skills that are explicitly exercised and discussed, which should lead to further learning success by the types of students currently having difficulty.

2. To address the limitations of the autograded feedback there are bi-weekly assignments graded manually . This should allow better quality high-level feedback to students, which should improve the quality of learning to those who pay attention to the feedback.
3. The course has switched from Maple to MATLAB as the programming language, to increase the potential of direct application in subsequent engineering work. Even though we believe that the primary value of a first course in programming is the transferable experiences about how programming languages and software design and development work, there may also be more enthusiasm by some students at using a language that is widely used professionally. The shift to MATLAB allows the course to more easily work with data analysis (e.g. interpretation of statistical results), data acquisition, and device control. This may broaden student interest and increase motivation. However, the course will continue in include some work on applications of mathematical modeling. Maple T.A. has the ability to execute MATLAB as a side process as well as the ability to execute modest amounts of MATLAB code by symbolically translating it and having the Maple engine execute it. Thus we feel that there are similar capabilities for autograding MATLAB work in Maple T.A. A simple MATLAB skill-building question is shown in Illustration 4.
4. The course is adding project-based learning assignments[22], allowing more open-ended and self-directed practice with programming, even though there is a continuing need for prefacing such work with smaller learning steps such as skill-building and "homework-sized" problem-solving[23] This should create circumstances for authentic student inventive thinking, potentially increasing student motivation to achieve success in programming.
5. In order to provide better quality automated program testing we also have introduced use of Cody Coursework™[24–26]. Cody Coursework automatically tests and scores student-submitted MATLAB programming. Test result output includes the same run-time error diagnostics that students see when running their programs on their own computers, addressing one of the most serious weaknesses of Maple T.A. used as a program tester. Cody Coursework is hosted in the cloud, so it removes the task of maintaining an autograding system from the university staff.
6. We are conducting experiments with using MATLAB autograding for badge awards. We believe that one of the drawbacks of courses such as this which strive to provide immediately useful skills to engineering students, is that it is difficult for "consumers of skill" such as employers or later courses to know exactly what the students should be able to do. We also are extending the use of autograded problems in a proctored proficiency exam to certification of specific MATLAB skills through an Open Badge-based certification scheme[27–30]. This will allow students to advertise their achievement of such skills in a verifiable and more detailed way than a course grade on a transcript. As an artifact, we intend to provide review documentation and on-line exercises so that one could attempt badge certification even after the course had ended.

As part of the course redesign an increased emphasis is being placed on systematic collection of information that can be used in formative evaluation of the course materials and pedagogical procedures. The goal is to guide further evolution of the course. This concern with metrics and measures includes ongoing development and refinement of measurement instruments that allow more complete understanding of the knowledge various students bring to their course and more

complete understanding of their views on such things as the usefulness of various course components in helping them to learn. With the course being run at scale these questionnaires are being developed to be administered on-line. The difficult thing here will be to engage enough students in the evaluation of course material to provide meaningful and useful data, To this end the formative evaluation data collection instruments are being informed by the work of others and by a process of design, evaluate and re-design intended to facilitate student engagement in providing feedback useful to course development. We also intend to analyze the autograding scores and interaction data to gain formative evaluation measures based on actual student work.

Conclusions

Our course attempts to provide first-year students with an experience that combines learning programming with using it for problem-solving and design in engineering, differing in emphasis and in problem selection from a generic CS1-style <<ref>> course. It uses autograding for a combination of proficiency-building, skill-assessment, and problem-based active learning. Autograding has become an important feature of the course, shifting use of human resources to face-to-face tutoring and higher-level formative and summative evaluation. We continue to explore the curriculum design space investigating the effects of additional time and staffing resources, additional varieties of computational engineering activities, project-based learning, and badge certification.

Acknowledgments

We wish to thank our colleagues Jeremy Johnson, Nagarajan Kagasamy, Baris Taskin, Pramod Abichandani, Frederick Chapman, David Augenblick, Mark Boady, John Novatnick, L.C. Meng, William Fligor, and Kyle Levin for their contributions to course's ongoing development. Part of the changes for 2013-14 were supported by a grant from Mathworks. Furthermore, the course has benefited from generous development support from the Department of Computer Science (now the Faculty of Computer Science) and the College of Engineering of Drexel University.

Bibliography

1. Sallee, T. *Synthesis of research that supports the principles of the CPM Educational Program*. (2005). at <http://www.cpm.org/pdfs/statistics/sallee_research.pdf>
2. Maplesoft. *Maple 17 User Manual*. (Maplesoft, 2013). at <http://www.maplesoft.com/view.aspx?SF=144530/UserManual.pdf>
3. Char, B. *Scripting and Programming for Technical Computation*. (Drexel University Department of Computer Science, 2012). at <https://www.cs.drexel.edu/complab/cs123/spring2012/lab_materials/Lab4/ScriptingAndProgramming.pdf>
4. Maplesoft. Featured Demonstration - Clickable Math. (2013). at <http://www.maplesoft.com/products/maple/demo/player/ClickableMath.aspx>
5. Char, B. & Augenblick, D. Scripting and Programming for Technical Computation - Lab Notes. *Scripting Program. Tech. Comput. -- Lab Notes* at <https://www.cs.drexel.edu/complab/cs123/spring2013/lab_materials/Lab3/LabsWithoutAnswersCS123Sp13.pdf>
6. Porter, L., Guzdial, M., McDowell, C. & Simon, B. Success in Introductory Programming: What Works? *Commun ACM* **56,** 34–36 (2013).
7. Herrmann, N., Popyack, J. L., Char, B. & Zoski, P. Assessment of a course redesign: introductory computer programming using online modules. *SIGCSE Bull* **36,** 66–70 (2004).
8. Anton, H., Blivens, I. & Davis, S. *Calculus, Early transcendentals*. (2010).
9. Klebanoff, A., Dawson, S. & Secrest, R. Batter Up: The Physics of Power in Baseball. (1993). at <http://umbracoprep.rose-hulman.edu/class/CalculusProbs/Problems/BATTERUP/BATTERUP_2_0_0.html>
10. Maplesoft. Maple T.A. - Web-based Testing and Assessment for Math Courses - Maplesoft. (2013). at <http://www.maplesoft.com/products/mapleta/>
11. Miller, L. D. *et al.* Evaluating the Use of Learning Objects in CS1. in *Proc. 42Nd ACM Tech. Symp. Comput. Sci. Educ.* 57–62 (ACM, 2011). doi:10.1145/1953163.1953183
12. Char, B. ASEE Webinar: Advanced Online Testing Solutions in a Freshman Engineering Computation Lab - Recorded Webinar - Maplesoft. (2013). at <http://www.maplesoft.com/webinars/recorded/featured.aspx?id=569>
13. Char, B. Developing questions for Maple TA using Maple library modules and non-mathematical computation. (2011). at <http://www.drexel.edu/~/media/Files/cs/techreports/DU-CS-11-04.ashx>
14. Hewett, T. T. & Porpora, D. V. A case study report on integrating statistics, problem-based learning and computerized data analysis. *Behav. Res. Methods Instrum. Comput.* **31,** 244–251 (1999).
15. Anderson, J. R., Conrad, F. G. & Corbett, A. T. Skill acquisition and the {LISP} tutor. *Cogn. Sci.* **13,** 467 – 505 (1989).
16. Rivers, K. & Koedinger, K. R. in *Intell. Tutoring Syst.* (Cerri, S. A., Clancey, W. J., Papadourakis, G. & Panourgia, K.) **7315,** 591–593 (Springer Berlin Heidelberg, 2012).
17. Rodrigo, M. M. T. & Baker, R. S. J. d. Coarse-grained detection of student frustration in an introductory programming course. in **Berkeley, CA, USA,** 75–80 (ACM, 2009).
18. Shaffer, S. C. & Rosson, M. B. Increasing Student Success by Modifying Course Delivery Based on Student Submission Data. *ACM Inroads* **4,** 81–86 (2013).
19. Chinowsky, P. S., Brown, H., Szajnman, A. & Realph, A. Developing knowledge landscapes through project-based learning. *J. Prof. Issues Eng. Educ. Pract.* **132,** 118–124 (2006).
20. Hewett, T. T. Cognitive aspects of project-based Civil Engineering education. in (2010). at <http://www.di3.drexel.edu/Orlando_PDFs/Hewett_Problem_Based_Final.pdf>

21. Detmer, R., Li, C., Dong, Z. & Hankins, J. Incorporating Real-world Projects in Teaching Computer Science Courses. in *Proc. 48th Annu. Southeast Reg. Conf.* 24:1–24:6 (ACM, 2010). doi:10.1145/1900008.1900042
22. Lehmann, M., Christensen, P., Du, X. & Thrane, M. Problem-oriented and project-based learning (POPBL) as an innovative learning strategy for sustainable development in engineering education. *Eur. J. Eng. Educ.* **33,** 283–295 (2008).
23. Boss, S. Project-Based Learning: A Case for Not Giving Up | Edutopia. at <http://www.edutopia.org/blog/project-based-learning-not-giving-up-suzie-boss>
24. Mathworks. MATLAB Cody Coursework. (2013). at <http://www.mathworks.com/academia/cody-coursework/index.html>
25. Mathworks. Cody Coursework for Instructors - MATLAB & Simulink. (2013). at <http://www.mathworks.com/help/coursework/cody-coursework-for-instructors.html>
26. Mathworks. About Cody. (2013). at <http://www.mathworks.com/matlabcentral/about/cody/>
27. Mozilla. Open Badges. *Open Badges* (2013). at <http://openbadges.org/>
28. Knight, E. *et al. MozillaWiki -- Mozilla Open Badges*. (2012). at <https://wiki.mozilla.org/Badges>
29. Carey, K. A Future Full of Badges. *Chron. High. Educ.* (2012). at <http://chronicle.com/article/A-Future-Full-of-Badges/131455/>
30. Higashi, R., Abramovich, S., Shoop, R. & Schunn, C. D. The Roles of Badges in the Computer Science Student Network. in **Madison, WI,** (ETC Press, 2012).