Creating autograded questions in technical courses

Bruce Char *

Abstract

We discuss our experience creating autograded questions for technical subjects. Scientific computation libraries, random variation, mathematical presentation, general-purpose programmability and symbolic computation are key ingredients to creation of questions that can be used for practice and evaluation in large technical courses. The variation is sufficient in some cases to make it likely that every student will receive a unique version of a problem. We have used these techniques the past four years in engineering classes including some involving 600-1000 first-year students. Over 120,000 student responses are automatically graded annually for homework assignments and proctored exams.

Keywords: Symbolic mathematical computation, test management systems, computer-based training.

Introduction

Web-delivery test management systems such as Maple TA[1] can invoke programs which use scientific computation libraries for question generation and automatic answer-checking and grading. They facilitate evaluation and practice on technical subjects for large classes of students. We have used Maple TA each term to between 500 and 900 engineering students each term in the past five years to give a series of eight out-of-class assignments, and a proctored in-class proficiency exam. Over 120,000 student questions are automatically graded annually for this class alone.

An important characteristic of the technology is its anytime-availability with immediate feedback. Successful exploitation of this allows scheduling exams and quizzes for large classes over multiple time periods, and allows unproctored student practice outside of class or lab. If the questions are static then students may find it easier to replace learning of facts or processes by learning by rote or Internet look up. This paper presents techniques for writing programs in Maple [2] that will present different variants of a problem to different students taking the same Maple TA assignment or test. These ideas could be used in any system and language providing comparable functionality.

The case for autograding

Deliberate practice is an crucial ingredient in learning [3, p. 236], particularly subjects that that require procedural knowledge. Giving students feedback is crucial to the development of their skill as technical problem-solvers, since they may not be efficient at telling how well their efforts are working. Recording evaluation results is useful in large courses where there are many details.

Conventionally instructional staff provide sufficient feedback using traditional “manual” methods of grading and interaction. Large classes of hundreds of students may required many hours of human-powered grading. This may become a significant component of cost course. Autograding systems may reduce this cost. It can make the difference between making grading feasible or infeasible.

Autograding systems can also provide immediate feedback on “practice whenever” exercises and tests which is infeasible for instructional staff. In our course, our end-of-term proficiency exam is advertised as asking questions seen in the term or that are on a list of extra problems made available in the week before

*Department of Computer Science, 3141 Chestnut Street, Drexel University, Philadelphia, PA 19104 bchar@cs.drexel.edu
the exam. Practice results are available 24/7 during the period before the exam week and outside of business hours during the exam week. In Fall 2011, our class of approximately 900 students took a total of 2867 non-credit, optional practice tests in preparation for the exam.

The present state of autograding can affirm that students have basic factual knowledge, or can solve straightforward problems. This is often a significant component of the content of many introductory courses with large enrollments. It can suggest that students may have weaknesses, and can automatically suggest hints and suggestions for remediation. We don’t view automatically graded (and recorded) feedback as a replacement human instruction or feedback. Rather, we see it as a way to staff effort to focus on situations where a deep understanding of the subject pedagogy can be used to coach students interactively.

Create questions that auto-generate variants

We have found that spending the extra time programming variation into a problem to be worth the effort, because it expands the useful lifetime of the question. With a static question, it may become well-known over time that the answer is (c), or “none of the above”, or 4.7. This is a problem if the same question is used over a period of time, as can happen in repeated offerings of a course over many terms, or in testing situations stretched out over multiple time periods as we face in our large course of 33 sections. Static questions must be rotated or varied each time they are given as exam questions. Static questions also make poor practice vehicles, since once the student has correctly answered them, no more learning or practice is required to answer the question again, it becomes just a matter of memory or lookup.1

Figures 1 - 6 show a question with automatically generated variants requiring numerical responses that we did for our Computation Lab I course in Maple TA. A short script adds variation to a word problem from a calculus book [4]. In this case the computation was intended to be Maple, but it could easily be a calculator or another kind of computation system. The original problem, which had fixed values for the various temperatures mentioned, was analyzed and the temperatures replaced by parametric names. Variant generation consists of using the built-in random number facilities to select integer values for the parameters.

Figure 1: A question with parametric variation, part 1

![Figure 1](image)

Figure 7 through 10 show a more extensive problem developed in Computation Lab II, after the students have already been introduced to how to fit data using linear least squares. Variant generation synthesizes a new set of data to be fit to a varying distribution. Figure 10 lists a portion of the Maple TA script generating

1If facile recall of facts is the point of the practice, then this may be fine. But if the object is mastery of procedural knowledge such as a problem-solving skill, then re-asking the question doesn’t supply more practice with the procedure, just the application of the procedure “look up the answer you had the last time”.

University of Massachusetts Lowell
April 27-28, 2012
the problem. This script invokes Maple programming written by the author and pre-loaded onto the quiz server, not shown. See [5] for the complete script and the Maple code.

Figure 11 shows a simple multiple choice problem. The variation there consists of choosing one response from several possible correct answers, and four responses from several possible incorrect responses. The variation comes from random selection rather than parametric variation.

Use the numerical libraries of your autograding system

Each variant potentially changes the following:

1. The parameteric values of the problem.

2. The description of the problem – both parameteric values but also wording (e.g. changing constraints, changing from singular to plural, changing “greater than” to “less than”).

3. The solution of the problem. These need to be given to the automatic grading component so that it can do answer checking.

Numerical solvers, data fitting, numerical evaluation, and execution of simulations are useful to handle variation automatically. Once the programming has been done to find the solution to a particular problem (which is probably a good idea to check things even if just creating a “one of” question to be administered the old-fashioned way with paper and pencil), it is straightforward to parameterize the calculation script to have it generate (and be able to automatically check answers to) thousands of variants.

Use technical presentation features of your autograding system

In any technical course, mathematical models may be used to describe problem scenarios, methods of solutions, or solutions. Changing the problem will also typically change the description of the problem to the student. While simple parametric variation may seem like it can be handled through a little string replacement, problem descriptions often include mathematical formulae and graphs. It is useful for the system to
be able to generate new versions of formulae and graphs. Figure 1 includes a call to the Maple procedure \textit{ExportPresentation}, which generates the MathML\cite{6} description of the parametrically altered variant of the formula that Maple TA (like some web browsers) renders in standard two dimensional math format. Figure 7 invokes Maple point and line plotting to display the generated variant of the data and the line.

Figure 12 describes a problem in a simulated car controller in which the presentation facilities are used to create an animated GIF to be viewed in the student’s web browser as part of the problem presentation. Students must write a control program that successfully navigate the car through several variant scenarios. The scenario outcomes are presented as animations similar to what the students should see when they run their control programs through the simulator. Creating the animations was so computationally extensive that it could not be done on-the-fly by the Maple TA server at test generation time. Rather, numerous scenarios were pre-computed and stored on disk, with dynamic variation occurring through random selection of a scenario.

\textbf{Use the symbolic computation features of your autograding system}

Figure 12 shows a differential equations problem from an engineering course on differential equations. The problem was generated by selecting from one of several expression templates for a particular solution, then choosing template parameter values so that modest amounts of work were needed for the student calculations.

Generating a solution and then working through the consequences to the answers and the presentation requires symbolic manipulation – use of programming libraries which know about the semantics of differentiation and of mathematical simplification and equivalence. For example, if one of the coefficients in a multi-term formula could be chosen as zero or one, one would want to make sure that the formula variant displayed had the appropriate simplification performed to remove vanishing terms. Answer-checking when formulas are required as the answer should be insensitive to the order that the student enters multiple terms of a multi-term expression, or if they enter an expression that is algebraically equivalent to what is given
to the answer-checker. Situations where variants of rational numbers or expressions are used should have greatest common divisors removed before display. Such functionality took a number of decades to be perfected for general-purpose use [7], but is now available commercially in systems such as Maple, Mathematica and Matlab, or open source in Sage[8]. Our experience with symbolic computation is that it is difficult to provide even a small part of this functionality in an ad hoc fashion with comparable power, reliability, or ease of maintenance.

Use the programmability of your autograding system

Figure 1 shows a problem that is handled through only a short script. However, for problems requiring a serious amount of student work, even more computation may need to be done by the question generator, since it must reproduce the answers expected of the student as well as setting up the problem variant and its presentation. In addition to mathematical libraries, we find that we use many features of general-purpose programming languages: operations on built-in of data structures (lists, arrays, queues, trees, etc.), string manipulation and word formatting (for creation of question statement variants).

Weighing the costs and benefits of autograding question development

We have found that it takes between half an hour to five hours or more to develop a question, depending on the complexity. This is comparable to the time reported independently in [9]). “Back of the envelope” comparisons of the time cost for human grading compared to the cost of question development and the
Figure 8: (a) Sensor problem plot (b) Sensor question

The first sensor in your collection had these readings:
\[ [1.000, 2.920], [8.000, 6.983], [13.000, 9.962], [30.000, 19.056], [32.000, 20.484], [36.000, 22.174], [38.000, 23.552] \]

(a) Fill in the blanks to create a Maple expression that creates the least squares fitted trend line for the data with independent variable a.
CurveFitting[ ]

(b) Fill in the blanks: the trend line formula is :

(c) What distance does the sensor read when it is placed 1.0000 centimeters from the target?

(d) What is the largest magnitude difference between the measured distance as given by the fitted line, and the actual distance in the range 4.0000 to 37.0000 centimeters? Enter a non-negative floating point number.

system expense lead one of the conclusion that the cost can be justified if the test items will be used enough times, for example for practice and examination in several offerings of a large course. However, we also note that the autograding systems take the same time to grade \( N \) variants of a problem as \( N \) identical versions, which would not be true for human grading.

Analysis of variants

A problem with use of random numbers to create problem variants is that the numbers create a version that makes sense. Naive creation of variants through random number generation can lead to problems. Variant creation must be constrained so that problems always

1. Have an unambiguous solution
2. State a problem that makes sense for the situation being modeled – no “A farmer has \( \sqrt{2} \) chickens....”.
3. Have solutions that makes sense – the solutions do not have negative mass, imaginary velocity, etc.
4. Have solutions that can be obtained by the student in a reasonable amount of time with the tools they are allowed to use.\(^2\)
5. Have solutions that can be automatically checked. If there is a unique solution, this is good. If there are several or many possible solutions, then the answer-checking programming should be good enough to accept any of them – or the problem’s directions rewritten to preclude those that can’t be recognized.
6. Have solutions that can’t be too easily guessed, unless guessing is an intended part of the learning. We have seen that in situations where many retries are allowed, some students try to substitute exhaustive enumeration for a more thoughtful approach. The problem and its solution must be formulated to make such “gaming” tactics unlikely to pay off.

For example, if one is using a quadratic formula \( a_2 \cdot m^2 + a_1 \cdot m + a_0 = c \) to describe a physics situation where the unknown \( m \) is a mass, choosing values \( a_0, a_1, a_2 \) and \( c \) at random could produce a version of the problem where the masses were negative or complex. We could do the analysis of how to pick \( a_1 \) and \( a_2 \) in a

\(^2\)We have discovered variants which have solutions which the student software can’t find, either due to ill-conditioning or limitations of the math library algorithms.
An alternative in this situation that often works, is to randomly select solutions and work backwards to get the starting formula. For a quadratic formula, the mathematics of avoiding negative roots is straightforward – pick two positive numbers $r_1$ and $r_2$ as roots, and use symbolic manipulation to reformat $(x - r_1) \cdot (x - r_2) = 0$ into whatever the desired presentation form for the quadratic.

Another possibility is to limit things to a large (i.e. more than the students can handle by rote or lookup) but tractable number of variants and exhaustively check them with the help of additional programming. Sometimes the easiest way to avoid asking nonsensical versions of the problem is to change the problem statement or the problem itself.
Software engineering

We have found that to deploy variant questions on a large course needs to be treated like a software engineering project. This includes:

1. Automatic testing. With non-trivial amounts of programming to generate variants, testing is a necessity, if only to explore all the possible branches that the variant generation code might take. As problems are modified due to design changes or bug fixing, re-testing is a necessity. The amount of work needed for this requires automated test scripts to be able to reproduce and hopefully check that the programming is functioning as required. Having scripts also means that regression testing can be performed so that fixed bugs do not reappear.

2. Testing can reduce the likelihood of a student finding a flaw in a question variant (improper presentation, unsolvable version of a problem, incorrect answer checking)? Over time we have moved to a policy where the presentation of the question includes the value of the random number seed. The seed can be chosen dynamically, but if the seed value is known then bug reports will allow reproducibility in further testing.

3. File repositories. Having everything in a code repository such as Subversion [11] means file sharing among multiple people or multiple computers, but it also makes it possible to retrieve older versions of files and withstand inadvertent deletion or alteration of the most current version. We think that this is a good idea with all course materials for all courses with a large staff.
4. With the time for testing built into the course development schedule, time for human quality assurance can also be scheduled. Our courses employ undergraduate lab assistants who also serve as question testers. For large courses, it is advantageous to know ahead of time that a question’s wording is easy to understand from several points of view, as well as whether students find the problems are do-able within the time available.

**Limitations**

The state of the art is far from ideal. There is neither extensive built-in knowledge nor a representation for the semantics of question creation as with [12, 13]. Anything reused between questions on different topics appears in an *ad hoc* form in the programs. We believe such features would reduce the cost and “learning curve” barriers of question creation.

Short-mathematical response questions provide only *inferential* evidence of learning (e.g. implementation, evaluation, synthesis as described in [14]). The interaction through the web browser of Maple TA does not allow it to observe what the student does on or outside of the desktop to get their answers. This makes it difficult to provide the adaptive feedback said to be needed for effective self-managed learning from Computer-Based Testing [15]. Even with this limitation, it can be used to as self-affirmation of learning or as a spur for questions to or consultations with staff.

Deeper interaction between software and student would seem to need software that can collect more information about what the student is trying and thinking about as they pursue answers. The Cognitive Tutor [16] approach provides automated adaptive instruction in subjects requiring procedural knowledge such as elementary technical problem-solving. Students work problems within an environment where the Tutor can observe the student working. It can certify that the student is following a reasonable solution process, or can provide context- and content-sensitive hints if the student is going down the wrong path.

The tutor uses production rules to describe the student’s problem solution process. If tutors varied problems using our techniques, the production rules would be altered only in situations where the solution algorithm is being varied. The tutor would need to dynamically load rules generated for the variant.

**Conclusions**

Auto-graded exercises can be enhanced for use in learning and evaluation by providing a collection of related problems rather than just a single problem. Some of the basic ideas for problem variation can be implemented in any system that provides suitable data structures and presentation preparation. For technical courses, robust algebraic manipulation functionality and mathematical calculation and presentation/formatting libraries make for convenient generation of variants. Variation incurs a higher cost in software engineering (analysis, testing, and design) than a typical one-paper/single-use assignment or exam question. However, the cost may be amortized by grading savings in several offerings over time of large courses. More sophisticated question variant generation awaits instructor-ready tools.

**References**


**Biographical Information**

Bruce Char is Professor of Computer Science in the College of Engineering at Drexel University. He is a past chair of ACM SIGSAM, the special interest group on symbolic computation. He has had a long-term interest in the application of IT tools to technical education.

**Acknowledgments**

The support of the Drexel University Department of Computer Science (Jeremy Johnson, head) is gratefully acknowledged. Mark Boady helped with question development and supplied the usage statistics.