Problem Generation Repository

Jason Nowell

July 9, 2023

Contents

1 Introduction

One of the most common requests from students is for more practice problems, and generating these problems can often be an exhausting process, even for relatively easy problem types. But coming up with problems that are simultaneously complex enough to provide good practice, and also whose solutions are numerically accessible and reasonable, can be challenging. For example, coming up with a polynomial such that the polynomial, along with its first and second derivatives, are all factorable over the rationals is a highly nontrivial task - let alone creating an assignment with a couple dozen of them.

One could search the internet for examples, but almost certainly any example on the internet includes a solution, which means the solution would be near the top of any Google Search list, and finding and verifying problems found on the internet is its own headache.

After tackling this problem myself, I started writing code to generate problems algorithmically, often leveraging advanced mathematical ideas to build elegant problems that suit my goals. Most online algorithmically generated problems tend to be brute force algorithms that simply randomize values ad-hoc, resulting in problems that, although technically accessible by the desired techniques, tend to result in messy values that require using calculators or approximating values by the students.

Take, as an example, the challenge of generating a factorable cubic polynomial. Most current online problem generators would generate problems by picking a random number in some interval and then plugging it into a generic form before handing the result to a student. So, if the random number is 5, it would plug it into a template of $x^3 - kx^2 + 4x - 4k$ because then the program knows that the correct factorable form is $(x - 2)(x + 2)(x - k)$.

In fairness, some more sophisticated problem generators will even allow a second random number to put in place of the 2 above, but the general method remains the same - and the worse online generators will just randomize the coefficients of the cubic and use some kind of discriminate type value to verify that three real solutions exist, without caring if those values are rational, irrational, or even reasonable to calculate.

In contrast, a much more elegant solution is to generate the three factors beforehand, then simply multiply them out and give them to the student - that is, to pick random integers a, b, and c, then multiply out $(x - a)(x - b)(x - c)$ to get the cubic for the student. Moreover, this approach allows for much more control over the complexity of the problem by the author. For example, you could instead multiply out something like $(a_1x - b_1)(a_2x - b_2)(a_3x - b_3)$ to allow for non-monic cubics, or you could simplify it down to $(x - a)(x + a)(x - b)$ to recreate the "good" online calculator, etc.

In the case of the cubic, it may not be too difficult to create the desired problems yourself, but some seemingly easy problems can become remarkably challenging to come up with nice algorithmic solutions to - or even to manually generate.

1.1 What does it mean to be a "Nice" problem?

When I say a problem is "nice" (good, elegant, etc) I really mean that it rates highly on the first two of my three aspects of problem writing - the third aspect is for the code that generates nice problems.

Pedagogically Useful:

Numerically Accessible: By numeric accessibility, I mean that the numbers students encounter and/or use for the problem are controllable to be relatively "nice" numbers - e.g. integers and/or rational numbers that are small enough to allow students to internalize and compute with them while maintaining some level of numeric intuition. In essence, a numerically accessible problem is a problem where all the numbers involved - from start to finish - are designed so that a student can maintain an understanding of their relative size and value on an intuitive level.

Although it is becoming more common to have random problem generation guarantee that a solution is a relatively "nice" number, very often this is the extent of numeric accessibility for these problem generators. Importantly, parts, or even the entirety, of the solution *process* may involve exceptionally opaque calculations. Consider, by way of example, the following problem:

Problem: Compute the definite integral: $\int\limits_{-2}^{2} \sin(3x) - \sqrt[3]{x} + 3x^2 \, \mathrm{dx} = 0$

On the one hand, the final answer to this integral is relatively nice - it's just 16. But, unless the student happens to notice that two of the terms are odd functions (and knows how to use this information in the context of integration) the step prior to this final "nice" answer will be all but indecipherable to the student:

$$-\cos(-6) - \frac{3}{4}(2)^{\frac{4}{3}} + 2^3 - \left(-\cos(6) - \frac{3}{4}(-2)^{\frac{4}{3}} - 2^3\right)$$

A truly numerically accessible problem maintains relatively nice numbers throughout the solution process, allowing for students to maintain decent numeric intuition throughout the process. Clearly this is subjective, and the nature of the question's intent also comes into play. For example, the previous integration question may actually be ideal in a section about the impact of even and odd functions in integration - allowing for an elegant demonstration of how much easier it is to use the even/odd property, rather than computing directly.

The goal then, is to have problems that are numerically accessible for students in the natural target audience for the problems. For example, for most precalculus problems there isn't much value in steps or answers that have combinations of irrational numbers - unless that is the specific point of the problem itself. Ideally, the code to generate problems will have corresponding comments that detail the steps and/or parameters selected to control the level of numeric accessibility - allowing for instructors to make changes if they have different pedagogical opinions than the original code author.

Algorithmically Reliable: It is nice to discuss the ideal properties of a problem - being numerically accessible and pedagogically useful, but if the code that generates the problems can't reliably generate problems that have these properties, then the problem generating code isn't really useful.

For example, if you want to generate a factorable polynomial, there are two obvious ways to go about it. You could generate a function of the form: $Ax^2 + Bx + C$ and randomly generate integers A, B, and C. This could, for the correct values of A, B, and C, generate a good problem - for example when $A = 1$, $B = 2$ and $C = -3$ you get a nicely factorable quadratic. Moreover, every problem generated this way is a quadratic, which means it is always factorable with the quadratic formula. But obviously you could get some pretty ugly problems too, like ones with irrational constants which require the quadratic formula. Or even worse, quadratics with non-real zeros. So, although this method of generation *can* generate good problems, it is not algorithmically reliable, because the result isn't necessarily a good problem. Indeed, the chances of getting a good problem is vanishingly small.

In contrast, you could achieve a much better result by generating the roots first, randomly generating four numbers A, B, C, and D, and then expanding the polynomial $(Ax - B)(Cx - D)$ (expansion can typically be handled by a command, rather than expanding it manually). This forces the function to not only be factorable with real zeros, but it also allows the author direct control over the zeros (you know the zeros are $\frac{B}{A}$ and $\frac{D}{C}$), which means you can print them in a solution manual or provide them to an online validation system - as well as having algorithmic control over the types of zeros you want to generate (for example, you could force A or C to equal 1 to get an integer solution rather than rational, or ensure that the zeros are the same or different by chancing the values against each other, etc). Most importantly, this generating method *always* generates nice problems, so it is algorithmically reliable.

1.2 The Solution - Crowd Sourcing

Thus the idea of this repository was born. After writing code that generated "nice" problems for a deep variety of precalculus and calculus problems, I realized that many others may have tackled these very same issues - and potentially come up with their own solutions to generating elegant problems. Moreover, there are probably a bunch of other people that are just starting their dive into this very same conundrum. Unfortunately, there doesn't seem to be any centralized repository of approaches to generating relatively "nice" practice problems.

1.3 About Me

My name is Dr. Jason Nowell. I got my PhD in mathematics from University of Florida in 2019, and I am currently the online content coordinator for the department of mathematics at University of Florida Online - which has been the top ranked online bachelor degree program for the last two years. I am also a developer for the Ximera Project, and the Florida affiliated version Xronos.

I have devoted most of my professional life to developing OER for online mathematics education, creating free resources for both instructors and students to drive down cost of education, while simultaneously increasing access and availability of quality mathematical tools for both. For a look at other projects and resources I am involved in, feel free to look at my personal website, which hosts tools and links to my various projects and resources for both students and instructors.

2 How to Use the Repository

3 How to Contribute