Teaching Floating-Point Representation with an 8-Bit Format: The One-Byte Float Approach

Dr. John K. Estell, Ohio Northern University

An active member of ASEE for over 30 years, Dr. John K. Estell is the Herbert F. Alter Endowed Chair of Engineering Sciences, and Professor of Computer Engineering and Computer Science, at Ohio Northern University. Dr. Estell is a nationally recognized leader in engineering education and accreditation. He currently chairs the ASEE Information Technology Committee and previously served on ASEE's Board of Directors as Chair of PIC III and Vice President of Professional Interest Councils. An ASEE Fellow, he has received numerous ASEE awards for his contributions to teaching, scholarship, and service. In accreditation, Dr. Estell has served as a commissioner for both the Computing and Engineering Accreditation Commissions of ABET, including roles on executive and training committees. He was named an ABET Fellow for his work in harmonizing practices across commissions. He also co-founded and serves as Vice President of The Pledge of the Computing Professional, promoting ethical standards in computing education.

Ian Meyer Kropp, Ohio Northern University
Dr. Stephany Coffman-Wolph, Ohio Northern University

Dr. Stephany Coffman-Wolph is an Assistant Professor at Ohio Northern University in the Department of Electrical, Computer Engineering, and Computer Science (ECCS). Previously, she worked at The University of Texas at Austin and West Virginia University Institute of Technology (WVU Tech). She is actively involved in community outreach with a goal of increasing the number of women in STEM and creating effective methods for introducing young children to CS concepts and topics. Dr. Coffman-Wolph's research interests include: Artificial Intelligence, Fuzzy Logic, Software Engineering, STEM Education, and Diversity and Inclusion within STEM.

Teaching Floating-Point Representation with an 8-Bit Format: The One-Byte Float Approach

Introduction

This paper addresses a topic that perplexes many first-year programming students: working with floating-point data. Throughout their primary and secondary schooling experiences, these students used real numbers in their STEM-related classes, where the number of digits is theoretically infinite and limited primarily via such concepts as significant digits. This changes in their introductory programming course (CS1), where an integer divided by an integer yields… an integer. And when computer-based floating-point values are stored or manipulated via computational expressions, the results sometimes differ from those in mathematical reality. Whether such differences arise from rounding, truncation, or irrational representation, students need to be aware of why they occur as the first step towards providing correct results using an imperfect system. However, springing the 64-bit version of the IEEE 754 Standard for Floating-Point Arithmetic on a bunch of beginning programmers tends to be an overwhelming, and overly technical, approach.

To the uninformed, it may seem unimportant for a software developer to understand the binary representation of floating-point numbers. For example, from a beginner C++ programmer's perspective, numbers can be represented as integers (e.g., the int data type) or as real numbers (e.g., the double or float data types). This programmer will likely assume that any real number can be represented as a double or a float type. However, this assumption is not only false but also dangerous in practice. Ubal et al. [1] found several disastrous floating-point-related bugs amongst a collection of software bugs collected by Thomas Huckle [2], including bugs that caused financial panics, misfired missiles, and errantly manufactured microprocessors. Clearly, this topic is misunderstood even by professional programmers, and their miscomprehension can have disastrous consequences.

In this study, we propose a novel method for teaching CS1 students the fundamentals of binary floating-point representation in an approachable way: The One-Byte Float. In the One-Byte Float, the instructor outlines how to design a floating-point number with 1 byte (8 consecutive zeros and/or ones) of memory. In doing so, students can learn about the components of a floating-point number without being overwhelmed by a 4- or 8-byte floating-point representation. This lesson is paired with an antecedent lab that forces students to wrestle with the limits of floating-point precision, thereby tying the concept to a relatable, real-world application [3]. We hypothesize that the lab and the One-Byte Float lecture are sufficient to increase the students' understanding of the basics of floating-point encoding, which we rigorously measure through student surveys.

Literature Review

Since its inception in 1985, educators have devised novel ways to teach the IEEE 754 standard to undergraduate students. Generally speaking, there are three broad methodologies used in pedagogy: 1) classroom exercises [4], [5], [6], [7]; 2) interactive software tools [1], [4], [6], [7],

[8], [9], [10]; and 3) simplified representations of IEEE 754 [1], [5], [6], [7]. First, we will provide a brief overview of IEEE 754, followed by a discussion of the exercises and the interactive software tools. Lastly, the simplified versions of IEEE 754 will be enumerated.

The IEEE 754 standard represents real numbers in the form $m \times B^E$(i.e., scientific notation), where m is a real number known as the mantissa, B is the base of the number (which we'll assume to be base 2), and E is the exponent. The mantissa has a decimal point following its first digit. The binary representation of a number in the format $m \times B^E$ is thus split into three fields: a one-bit sign for the mantissa, a binary sequence for the mantissa, and a binary sequence for the exponent.

The earliest examples of floating-point exercises include Scott [5], who devised a series of pen-and-paper exercises using a simplified 16-bit floating-point standard. The goal of the 8 exercises is to give students hands-on experience with a floating-point standard, while remaining simpler to handle by hand than a 32- or 64-bit floating-point representation. Examples of these problems are conversions, boundary calculations, and reflection questions. Michael Overton's book *Numerical Computing and the IEEE Floating Point Standard* [6] included similar comprehension questions after each section. However, the length and relative complexity of the questions make them more suited for at-home studying. Fernadez et al. [7] use a web tool to support three categories of exercises which explore 1) the effects of truncation modes, 2) the relationship between range and precision in different floating-point representations, 3) the procedure of arithmetic operations, and 4) the nature of various algebraic anomalies.

There are numerous interactive tools for floating-point education available across a wide range of platforms and purposes. The functionalities include design tools for custom floating-point numbers (e.g., number of bits in the mantissa or exponent) [1], [6], illustrated arithmetic operations [1], [4], [7], [8], and quizzes [9]. Most of these tools are web applications [4], [8], [9], while Ubal [1] and Overton [6] developed desktop and command line applications, respectively.

Finally, and most relevant to this work, are simplified representations of floating-point standards with few bits, often referred to as minifloats [11]. Since the definition of minifloats can vary between publications, we will assume these minifloats have 16 bits or less, since single-precision floats in IEEE 754 contain 32 bits. Minifloats have applications both in industry and pedagogy. Recently, minifloats have seen renewed interest in industry for machine learning [12]. From an engineering education perspective, these smaller floats are easier to work with than 32- or 64-bit floats, which are difficult to visualize and calculate by hand [5].

Next, we will enumerate the educational minifloats used in the literature. Scott [5] designed an educational 16-bit floating-point system with 1 sign bit, 5 exponent bits, and a 10-bit mantissa. This system included custom arithmetic algorithms, making the format easier to work with than the IEEE 754 format. In an exercise in his textbook, Overton 2001, made a brief and very simple 4-bit toy floating point system used alongside a comprehension problem. This system was less fleshed out than Scott [5], but was notable for its size. Fernandez et al. [7] developed a simplified 12-bit floating-point representation: 1 bit for the sign, 5 bits for the exponent, and 6 bits for the mantissa. Unlike Scott [5], Fernandez et al. also designed the floating-point system to be expandable. This way, students can experiment with the design of floating-point representations

and gain a deeper understanding of the trade-offs involved. This format is supported by an interactive online tool. Finally, Ubal [1] doesn't have a single simplified floating-point representation; instead, it uses exercises in which students use an interactive desktop application to design custom floating-point formats. This application allows students to design floats as small as 2 bits (1 bit exponent, 1 bit mantissa), though, as one would expect, the format is unusably limited.

To the best of our knowledge, there is a gap in the literature for an in-class tool to teach a simplified IEEE 754 format, supported by survey evidence. There are studies that proposed in-class exercises, studies that proposed simplified forms of 754, and studies that used surveying for validation, but none that combined all of these features. Also, none of the studies used an 8-bit simplified floating-point format, which we hypothesize is a unique sweet spot not covered in the literature.

Pedagogical Design

By limiting the representation to just 8 bits, the One-Byte Float dramatically reduces both range and precision, but gains a critical pedagogical advantage: simplicity. Values range from 0.03125 to 7.875, the least significant mantissa bit involves a weight of just $1/64^{th}$, and all 256 possible values can be printed out for examination. With this format, students are able to examine how floating-point values are implemented as unequally distributed discrete values along the real number line, how some values in base-10 (such as the fraction $1/10^{th}$) cannot be converted to an exact base-2 representation, and what would need to be changed in order for this data type to represent zero, include negative numbers, and so forth. While it's not expected that anyone would want to use One-Byte Float as a data type in an actual program, it allows for meaningful class discussions by making all values and concepts approachable.

As course instructors, we have to deal with the pedagogical tensions around when to provide direct instruction versus when to foster productive struggle. Productive struggle, perhaps better expressed by an athlete's "no pain, no gain" mantra, is often encountered when designing a solution to a real problem, such as calculating sales tax. This establishes a more relevant need, motivating students to be curious about the design space and actively seek appropriate solutions. However, some students can get easily frustrated, while others will feel that their time was wasted once things are explained. The direct instruction approach provides scaffolded learning, with all in the course starting from a similar foundational basis for approaching the problem. However, providing the answer early on can lead to a shallow understanding of the underlying issue, thereby missing a learning opportunity. The attempt here was to design a structured discovery approach that charts a middle path. Laboratory assignments that focused solely on performing sales tax calculations and making change from a cash transaction were designed as opportunities for productive struggle, with the safety net of instructors and teaching assistants providing guidance whenever students encountered difficulties.

The One-Byte Float concept was introduced in a lecture following the first major assignment, which was to calculate the cost of a transaction at a donut shop. Such transactions generally include discounts for purchasing a dozen doughnuts, different price points for doughnut types, calculation of sales tax, and making change for cash transactions. Laboratory assignments were

used to break down the various tasks, allowing students to focus on these separate but interrelated problems. Thus, students had separate problems that focused solely on calculating the sales tax as specified by the Ohio Department of Taxation and on making change at the end of a transaction. In all of these cases, there exists a hidden complication: many of the fractional values involved, such as 0.01, cannot be accurately represented in base-2 floating-point, as their binary values endlessly repeat, like the fractions ⅓ and 1/7 when decimalized in base-10 representation. Thus, students must contend with this issue - an important one, given that it is a financial transaction - as they write their programs.

The format of One-Byte Float is shown in Figure 1. We used a switch-based approach to make the concepts of ones and zeros more approachable as many understand the binary affordance of a switch: a 1 represents turning something on, and a 0 represents turning it off..

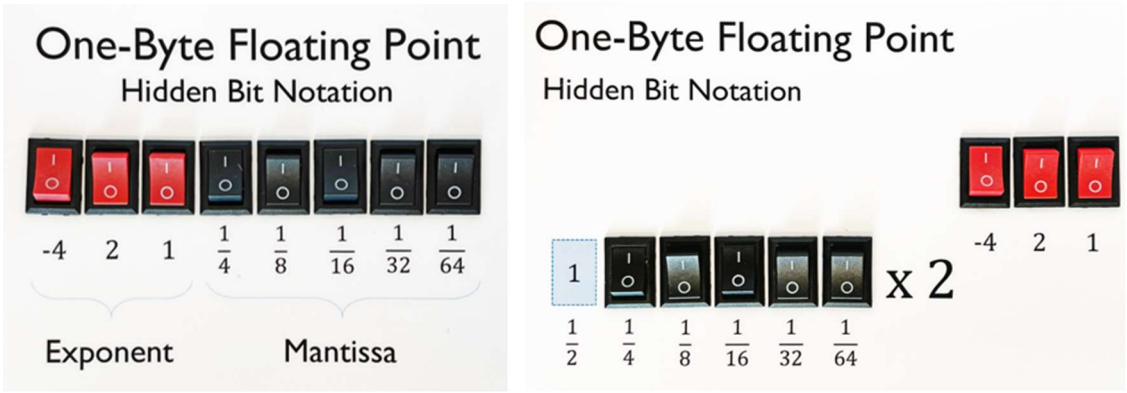

FIGURE 1. TWO REPRESENTATIONS OF ONE-BYTE FLOAT.

The five black switches represent the mantissa bits; the three red switches represent the exponent. Two images are provided to illustrate different aspects of this representation. The left image shows how the byte would be stored in memory. The second image (on the right) shows the actual representation of the value, including two important features. First, it visualizes the "hidden bit," a mathematical optimization that works only with binary values. This optimization is possible because normalizing a binary value always yields a 1 as its leading digit. Second, it positions the mantissa bits, with the leading hidden bit, and the exponent bits where one expects to find them. Determining the represented value is straightforward: if the bit is set to one, then you include the weight associated with that bit; otherwise, if the bit is set to zero, you don't. Thus, for the bit pattern 10010100 shown in Figure 1, the One-Byte Float value is:

$$\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{16}\right) \times 2^{-4} \;=\; \frac{52}{64} \times 2^{-4} \;=\; 0.05078125$$

Methodology

Research Question

The research question for this paper is as follows:

> **RQ1.** To what extent does the One-Byte Float representation improve students' conceptual understanding of floating-point arithmetic?

We hypothesize that a positive effect will occur, as with only 256 possible floating-point values, and with all values contained within the 0-to-8 range, students can literally print out and examine the entire representation space. In comparison, the IEEE 754-2019 Standard for Floating-Point Arithmetic contains 2^{64} values, with an extreme exponent range ($10^{\pm 308}$) that is beyond human comprehension. By reducing cognitive load, students can focus on understanding the underlying concepts rather than being overwhelmed by the sheer complexity of standard floating-point representations.

In accordance with the Department of Health and Human Services' Regulations for Protection of Human Subjects (45 CFR 46.110), the Institutional Review Board (IRB) at Ohio Northern University reviewed the protocol associated with this research (JE-EN-090425-1) and granted it exempt review status, under exempt category #2 – Tests, surveys, or observation of public behavior.

Participants

A total of 48 students across two sections were enrolled in the Fall 2025 offering of the Introductory Programming course. Regarding declared majors, 17 identified as computer engineering and 11 identified as computer science. Of the remainder, two are mechanical engineering students pursuing a computer science minor, while the 18 other students come from majors such as chemistry, electrical engineering, molecular biology, and physics, all of whom are taking Introductory Programming as either a required course for their major or to satisfy a general education requirement for their major. Of the 48 enrolled students, 47 completed the survey (98% participation rate). Of the 28 students identifying as either computer engineering or computer science majors, 27 completed the survey (96% participation rate), and 26 persisted to the Spring 2026 offering of the second introductory course, Object-Oriented Programming.

Methodology

The original research design called for students to complete both pre- and post-activity surveys based on the authors' prior CS1 pedagogical research covering all activities associated with the first major programming assignment, with matching Likert-scale questions analyzed using a dependent-samples *t*-test to determine statistical significance (*p*-value) and effect size (using Cohen's *d* classification). However, due to one author's health issues, only the post-activity survey was modified by adding two Likert-scale questions specific to the One-Byte Float lecture and subsequently administered. This survey included four demographic questions, 32 quantitative questions using a 7-point Likert scale ranging from "strongly agree" to "strongly disagree," and four qualitative questions.

Results

While the computer engineering and computer science majors are first-year students, there are many sophomores and juniors, plus the occasional senior, amongst the remaining students. Consequently, a population roughly split down the middle between first-year students taking their required introductory programming course and students from a range of majors and academic levels makes it difficult to evaluate the assessment data. The best approach is to look at

the class as a whole and separate out the data for first-year computer engineering and computer science majors, herein referred to as "majors" in the text.

The first question to be addressed is, "The One-Byte Float lecture helped me better understand the limitations of representing floating-point values in a computer." The 7-point Likert-scale responses are shown as a histogram in Figure 2, with "Strongly Disagree" corresponding to 1, and "Strongly Agree" corresponding to 7. The y-axis refers to the number of responses.

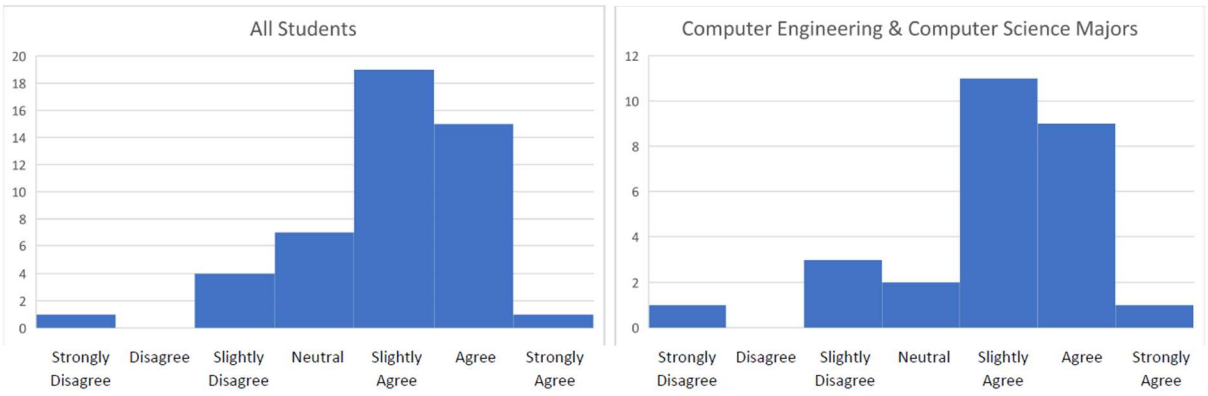

FIGURE 2. RESPONSES TO THE QUESTION, "THE ONE-BYTE FLOAT LECTURE HELPED ME BETTER UNDERSTAND THE LIMITATIONS OF REPRESENTING FLOATING-POINT VALUES IN A COMPUTER."

Numerically, the overall mean of the response was 4.96 (approximately "Slightly Agree"), with a standard deviation of 1.11, while for the majors, there was an identical value for the mean, with a slightly larger standard deviation of 1.26. Collectively, this implies that the students showed a mildly positive response, with reasonable agreement amongst the respondents. In other words, the results were promising but could be improved. The next question being examined was added as a control: "The One-Byte Float lecture helped me better understand the limitations of representing integer values in a computer." This lecture had no content whatsoever on integer representation; accordingly, one would expect significant disagreement with this statement. However, as Figure 3 illustrates, this was not exactly the case.

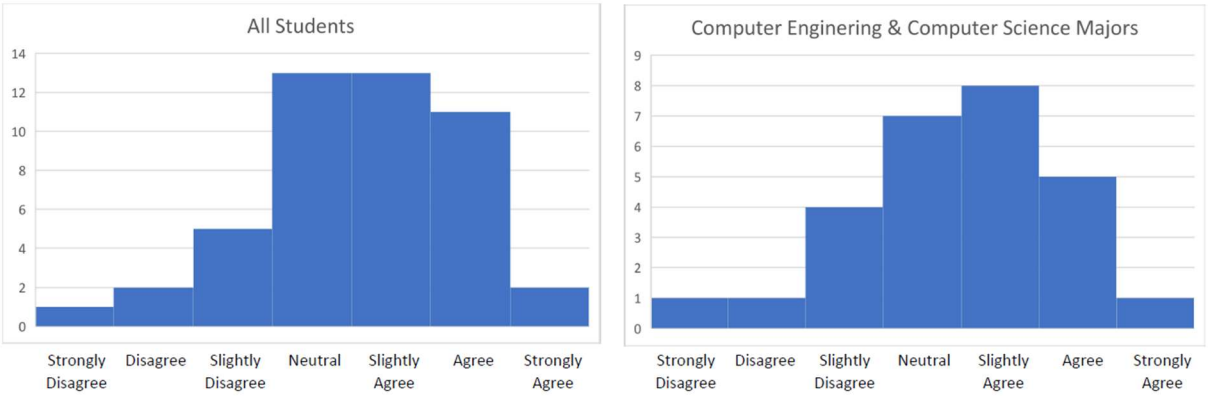

FIGURE 3. RESPONSES TO THE QUESTION, "THE ONE-BYTE FLOAT LECTURE HELPED ME BETTER UNDERSTAND THE LIMITATIONS OF REPRESENTING INTEGER VALUES IN A COMPUTER."

For all students, the mean response was 4.62 (approximately "Slightly Agree"), with a standard deviation of 1.30. For the majors, the mean was 4.44, and the standard deviation was 1.26. There are multiple possibilities for this result. First, it is possible that the students overgeneralized: while the lecture conveyed the idea that computers have limitations in their ability to represent numbers, students may have conflated data types, incorrectly assuming that the same limitations associated with floating-point representation, such as precision and range, also apply to integers. It is also possible that, at this point in the survey, students might have experienced survey fatigue and therefore not read the question carefully.

To investigate which explanation best accounts for this unexpected result, a secondary analysis examined the numerical difference between each student's Likert-scale responses to the floating-point and integer questions. Ideally, this should be strongly positive, given that the lecture focused solely on floating-point representations. Figure 4 shows the distribution of these differences.

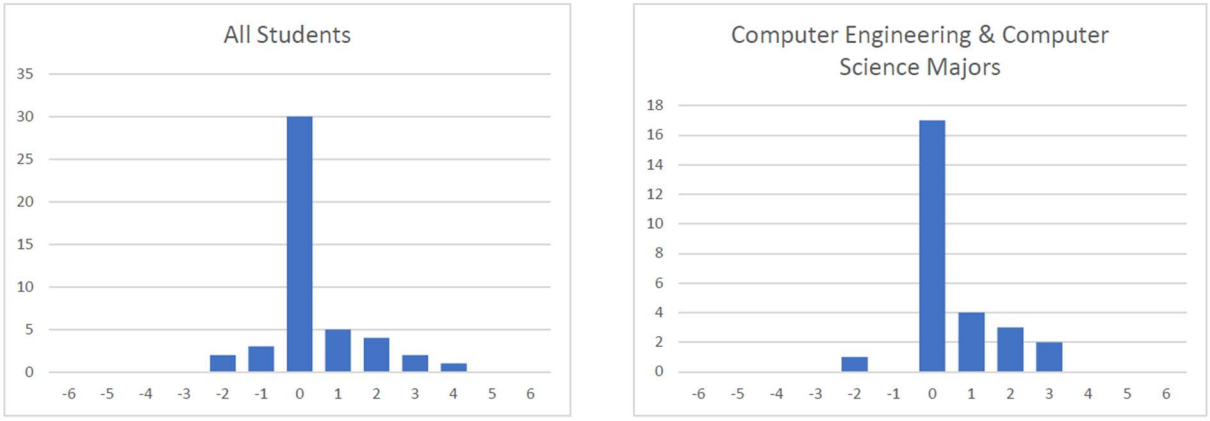

FIGURE 4. DISTRIBUTION OF DIFFERENCE SCORES (FLOATING-POINT RESPONSE MINUS INTEGER RESPONSE). POSITIVE VALUES INDICATE STUDENTS RATED THE LECTURE AS MORE HELPFUL FOR UNDERSTANDING FLOATING-POINT LIMITATIONS THAN INTEGER LIMITATIONS.

Overall, 30 out of the 47 students in the course (64%), and 17 out of 27 majors (63%), gave identical responses to both the floating-point and integer questions. This difference distribution is roughly normalized, with a mean close to zero (all: 0.34; majors: 0.52), indicating that students were not adequately discerning the difference between the two questions. It is worth noting that both datasets exhibit positive skewness, and the dataset for the majors has almost no negative tail. Therefore, some students (33% of the majors, 26% overall) recognized that the lecture was more relevant to floating-point representation.

There is one additional factor that might explain why students conflated integer and floating-point concepts: the lack of explicit comparisons in both the lecture and subsequent reinforcement activities (i.e., homework and laboratory assignments). While the instructors designed an assignment sequence in which students would have laboratory assignments on real-world floating-point issues, such as sales tax and making change calculations, there was no assignment or any related follow-up activity to reinforce the One-Byte Float material. Without an activity that explicitly contrasted integer and floating-point representations, students may have viewed the One-Byte Float lecture more broadly as addressing numeric representations in computers

than as specifically addressing floating-point representation. Having nearly two-thirds of the class providing identical responses to both questions suggests that they failed to cognitively separate these two representation concepts.

Limitations

A critical issue with the survey is the large number of quantitative questions, which can easily lead to survey fatigue. Also, while the demographic data was collected, our analysis did not account for a key demographic variable: each student's prior programming experience. Explaining a concept that is already understood, based on prior knowledge, can skew the results. Acquiescence bias - a tendency to agree with positively worded statements - could have skewed the data reported for the integer value question. Most critically, this study measured only student perceptions of learning through self-reported surveys. We did not directly assess conceptual understanding through knowledge checks, error analysis, or longitudinal tracking of student performance on floating-point problems. Future work will include objective measures to validate whether perceived understanding translates to actual competence.

Next Steps

While, at first glance, the results of this research are less than stellar, several "diamonds in the rough" have emerged that, when polished, will hopefully improve students' understanding of the nuances related to computer-based data representation methods.

Course and assignment modifications

Unfortunately, several students failed to see the connection between those lab assignments and the overall problem of handling all aspects of a sales transaction, to the point of writing a completely different algorithm instead of just copying and pasting an already validated solution into the appropriate location in the final program. Taking more time to explain, prior to the lab, that there will be "unexpected" (to them) rounding behavior and that we will address it in the first lecture following that lab may reduce frustration while both normalizing the struggle and preserving the joy of knowledge discovery by not giving away the solution. Additionally, the lab assignments can be modified to include additional checkpoints that help steer students toward the primary representation issue: being off by a penny when calculating what change is owed. This can be further addressed by having students attempt a simplified version of the making change calculation: first using an integer representation (which may seem counterintuitive but can be thought of as a pile of pennies) and then with the "obvious" double-precision floating-point representation, thereby highlighting the difference. The One-Byte Float lecture would then be more relevant given the planned shared struggle. Adding post-lecture assignments where students manually convert base-10 floating point values into One-Byte Float representation - and vice versa - would help to reinforce the concepts involved.

Assessment modifications

The foremost improvement would be to craft a much shorter survey focused on floating-point representation, rather than broadly examining the set of assignments currently in use for the first major programming project. Such a survey would alleviate survey fatigue. The current set of

qualitative questions does not specifically address floating-point representation; it should include targeted questions for this learning outcome and exclude all other types of queries. Questions can be made more specific by providing additional information or citing examples, thereby requiring students to discern that the two questions differ by only a single word or phrase. As for the quantitative questions, adding an exclusionary option, such as "not applicable" or "not covered," would allow students who realize a particular topic was not covered to provide a more appropriate response. Finally, given that Likert-scale questions are only measures of perception, adding some quiz-like questions would allow measurement of one's understanding of these concepts, which can then be compared with responses to other survey questions.

Reinforcement elsewhere

Revisiting the One-Byte Float format is readily achievable in Object-Oriented Programming, our second-semester introductory programming course. This course uses Java and has as two of its outcomes the ability to develop graphical user interfaces and employ event-driven programming. Accordingly, students can be assigned to develop an application that implements One-Byte Float by visually showing all 8 bits of the byte in question, allowing users to toggle each bit's value between 0 and 1, showing the bit's interpretation and weight, and displaying the resultant base-10 interpretation. Data representation concepts can be further reinforced by allowing the byte to toggle between integer and One-Byte Float representations, specifying whether the mantissa is signed or unsigned, and incorporating the representation of zero.

Conclusion

A fundamental purpose of educational research is to determine what works. However, equally important is identifying what does not work as intended, as these findings can drive meaningful improvement. Although the One-Byte Float approach showed promise, with students indicating moderate agreement that it improved their learning, nearly two-thirds conflated integer and floating-point concepts, revealing a gap in their understanding. The evaluation of the resultant data led to actionable insights, with planned modifications to both the assignment and its assessment instrument, and the development of a reinforcement activity for use in the subsequent programming course for the majors. By questioning how well our instructional practices work and continuously seeking improvement through an iterative, evidence-based cycle of implementation, assessment, and refinement, we can better achieve our primary goal: helping students learn.

References

[1] R. Ubal, J.-C. Cano, S. Petit, and J. Sahuquillo, "RACFP: a training tool to work with floating-point representation, algorithms, and circuits in undergraduate courses," *IEEE Transactions on Education*, vol. 49, no. 3, pp. 321–331, 2006.

[2] T. Huckle, "Collection of software bugs," *Institut für Informatik TU München: Munich, Germany. Available online: http://www5. in. tum. de/∼ huckle/bugse. html (accessed on 7 January 2026)*, 2004.

[3] J. K. Estell, S. Coffman-Wolph, and I. M. Kropp, "'Mmm… Donuts!' Using Real-World Scenarios in a First-Year Programming Course," in *2023 ASEE Annual Conference & Exposition*, 2023.

[4] I. Koren, *Computer arithmetic algorithms*. AK Peters/CRC Press, 2018.
[5] T. J. Scott, “Mathematics and computer science at odds over real numbers,” *ACM SIGCSE Bulletin*, vol. 23, no. 1, pp. 130–139, 1991.
[6] M. Overton, “Numerical Computing and the IEEE Floating Point Standard,” *SIAM*, vol. 270, pp. 271–274, 2001.
[7] J.-J. Fernández, I. Garcıa, and E. M. Garzón, “Floating point arithmetic teaching for computational science,” *Future Generation Computer Systems*, vol. 19, no. 8, pp. 1321–1334, 2003.
[8] E. Weitz, “IEEE 754 Calculator,” IEEE 754 Calculator. [Online]. Available: https://weitz.de/ieee/
[9] B. Peng, “Online Tutorial Tools to Practice Data Representation,” in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*, 2022, pp. 990–995.
[10] H. Schmidt, “IEEE-754 floating point converter.” [Online]. Available: https://www.h-schmidt.net/FloatConverter/IEEE754.html
[11] S. Aggarwal *et al.*, “Shedding the bits: Pushing the boundaries of quantization with minifloats on FPGAs,” in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2024, pp. 297–303.
[12] S. Narasimhan, “NVIDIA, arm, and intel publish FP8 specification for standardization as an interchange format for AI,” *Nvidia technical blog, September 14th*, *2022*.