by Henry M. Walker,
Grinnell College

Teaching Binary Representation of Numbers, Highlighting Payoffs for Students Interested in Software

Traditionally, undergraduate computing curricula have included discussion of the binary representation of numbers (and other data types). Often this general topic may be included in courses on Discrete Structures/Mathematics or Computer Organization/Architecture, although numerous courses may touch on this material. However, as discussed in Section 1, curricular recommendations and practices may vary considerably. Further, discussions may have numerous foci, such as digital design, underlying mathematical properties, implications for software development, etc.

Although this column mentions several foci, it highlights topics and considerations that seem most relevant for students planning careers related to programming and software development. Subsequent sections include:

2. Simple Arithmetic with Integers
3. Basic properties of floating-point numbers and casting
4. Boolean expressions and loop conditions with floating-point numbers
5. Error Accumulation During Addition of Floating-point Numbers

The column concludes with Acknowledgements, Conclusions and References. Most sections include both a short discussion and simple examples that may be useful within a class setting.[1]

[1] This column complements my February 2022 column, "what is the Payoff?" [12]. The earlier column suggests four possible, high-level principles for payoff, whereas this column focuses specifically on binary representation and potential examples and/or exercises.

For some readers, much or all of this material may be well-known and part of existing courses. For others, several topics may not be covered currently; for example, Section 3 may or may not be discussed, or perhaps only abstractly. Comments from reviewers of early drafts of this column describe both extremes, as do recent discussions with faculty currently covering this material in one or more courses. One goal of this column is to collect reasonably simple examples/exercises to highlight ways to discuss consequences of binary representation for programming—even without the need for extensive, detailed discussions of representational details.

1. Curricular Recommendations, Resources, Practices, and Observations

Discussion regarding the coverage of data representation for numbers naturally begins with professional curricular recommendations, and then moves to common resources, practices, and thoughts looking ahead.

ACM/IEEE-CS Recommendations: *Computing Curricula 2023* specifies 3 Core-Tier2 hours on the Machine Level Representation of Data, with the following topics and outcomes [3, pp. 63-64].

TOPICS:

- Bits, bytes, and words
- Numeric data representation and number bases
- Fixed- and floating-point systems
- Signed and floating-point systems
- . . .

LEARNING OUTCOMES:

1. . . .
2. Explain the reasons for using alternative formats to represent numerical data [Familiarity]
3. Describe how negative integers are stored in sign-magnitude and twos-complement representations. [Familiarity]
4. Describe how fixed-length number representations affect accuracy and precision [Familiarity]
5. . . .
6. Convert numerical data from one format to another [Usage]
7. . . .

Similar lists appeared in CS2001 [2, p. 98], although the "Familiarity" and "Usage" designations were not used before 2013. Interestingly, CS2013 recommends greater mastery for conversion of formats than consequences related to accuracy and precision. More recently, a similar listing is included in the draft CS2023-Gamma recommendations [4, p. 57], but only one hour is designated, and consequences of data representation are barely suggested. Altogether, professional recommendations expect a modest level of coverage of the binary representation of numbers, but time allocation seems too short to expect mastery or a clear understanding of consequences for programming.

Resources: For instructors wishing to discuss numeric data representation in detail, numerous sources are available, including the following.

- Hennessey and Patterson integrate matters of data representation with

discussions of circuit design [6]. This series of texts provides considerable technical detail, with a focus on underlying issues and algorithms.
- Overton [7] focuses on floating-point numbers and underlying mathematical concepts and issues.
- Goldberg [5] highlights elements for floating-point numbers from a reasonably formal mathematical perspective.

Some schools, particularly certain liberal arts colleges, may condense foundational topics in architecture and operating systems into a single course, leaving room in an undergraduate program for other topics that seem more relevant to their students.

Both books [6,7] provide considerable details regarding the data representation of floating-point numbers. For example, both discuss four approaches in the IEEE standard for "correctly rounded values" when storing real numbers that may not have exact floating-point representations. In addition, both include wonderful examples of historical circumstances that tie data representation to real-world events/consequences. However, with such detail, these treatments are quite extensive (e.g., Overton's discussion of floating-point formats, techniques and consequences, while reasonably accessible/readable for students, extends beyond 100 pages). Goldberg is much shorter, but may seem too abstract, with too much mathematical notation, to be accessible for some [many?] student groups.

Some other books devote substantially less space to binary representation, with much less discussion of the consequences of these data formats. Pragmatically, these books may reflect the time constraints from the ACM/IEEE-CS recommendations.

Practices: Overall, discussions with faculty and a review of faculty comments and resources suggest that instructors vary substantially on the amount of time devoted to numeric data representation and consequences within courses. (For example, as mentioned above, reviewers of an early draft of this column were clearly split on their past practices.)

At a basic level, the amount of coverage may reflect faculty and departmental decisions concerning priorities. Based on those priorities within a time-limited curriculum, some computing faculty may devote substantial class time to these topics (well beyond the recommended 3 hours), students may convert integers and floating-point numbers from decimal to binary and back, and discussions may emphasize practical consequences.

On the other hand, some faculty may emphasize other topics, some discussion may include limitations of numeric representation (overflow for integers, accuracy considerations for floating-point numbers, etc.), and students may gain little direct experience with these consequences.

As a separate matter, some schools, particularly certain liberal arts colleges, may condense foundational topics in architecture and operating systems into a single course, leaving room in an undergraduate program for other topics that seem more relevant to their students. In such degree programs, students may gain considerable background and insight into areas related to careers in their geographic region, such as cybersecurity, software development, algorithms, artificial intelligence, mobile computing, etc. However, considerations of binary representation may be minimal.

In addition, even when these topics are discussed earlier (perhaps even in two lower-level computing courses, students may not think about basic consequences of data representation in a later course. For example, after observing that many of students (sometimes 80%–90%) seemed to ignore such matters in upper-level courses, a faculty member at one school wondered whether students missed this topic in earlier courses, failed to understand the significance of data representation in computations, chose to take shortcuts, or otherwise decided to ignore these matters in their programming.

Thoughts Looking Ahead: For faculty who already devote considerable time to both the details of data representation for numbers and consequences, the following parts of this column may provide generally interesting examples, but skimming what follows may seem appropriate.

For others, two separate thoughts may be useful.

- Payoff: When devoting time to a topic within a course, a natural question involves what payoff might be derived. In this case, if students devote effort to learning about the binary representation of numbers, how might this impact their work and insights in the future? For those interested in hardware and circuit design, connections to data formats may have direct relevance, and numeric formats can provide solid explanations of program behaviors. However, for those interested in careers in programming and software development, payoff can (and likely should) be connected explicitly to the writing of I/O, conditionals, and loops.
- Timely Curricular/Course Review: After being reminded of the impact of binary representation on the writing of programs, several faculty have remarked, in effect, "I have not devoted much time to these consequences in the past, but this conversation/reading reminds me that I should adjust my emphases in the future."

The remainder of this column considers how the binary representation of numbers can impact program behavior and correctness in at least four ways:

- simple arithmetic (e.g., finding averages) with integers,
- basic properties of floating-point numbers and casting,
- Boolean expressions and loop conditions with floating-point numbers, and
- error accumulation during addition of floating-point numbers.

The following sections examine these areas and suggest possible examples that might be used in class or in assignments.

2. Simple Arithmetic with Integers

Many students may understand that many popular programming languages place limits on the size of integers, although some do not. More commonly, in my experience, students know conceptually that, within range limits, integers are stored exactly, and processing is exact (subject to possible truncation during division). However, since students typically work only with rather small numeric values, they may not experience potential issues arising from overflow, and thus they may forget about the potential for overflow when writing or reviewing programs.

As a result, the following two examples/exercises can lead to worthwhile class discussions.

EXAMPLE 1: FINDING THE AVERAGE OF TWO NUMBERS.

Given two integers, `i` and `j`, a program is supposed to compute their average (as an integer). (Here, if the arithmetic average is a real number ending in .5, the average may be rounded either up or down. Thus, either 7 or 8 should be considered as a correct [integer] average of 6 and 9.)

Five approaches are proposed:

```
avg1 = (i + j) / 2;
avg2 = i/2 + j/2;
avg3 = (i+1)/2 + j/2;
avg4 = (i+1)/2 + (j+1)/2;
avg5 = i + (i-j)/2;
```

a. Which, if any, of these approaches will work reliably for all (or almost all) non-negative integers?

b. Which, if any, of these approaches will work reliably for all (or almost all) integers (positive, negative, or zero)?

In my experience, many students do not consider potential issues of overflow (e.g., for `avg1`), and they often choose `avg1` as their preferred approach. Interestingly, as readers may know, approaches 3 and 5 can work without trouble for non-negative integers (except approach 3 encounters overflow when i is the maximum possible integer), but all of these approaches have potential issues when all integers are allowed.

EXAMPLE 2: AVERAGING AN ARRAY OF INTEGERS:

Consider the following C code to average 10 numbers in an integer array:

```
int i;
int sum = 0;
for (i= 0; i < 10; i++) {
 sum += arr[i];
}
intAvg = sum / 10;
doubleAvg = sum / 10.0;
```

Students may be asked to run this code on a machine with 32-bit integers, where the range of integers is -2147483648 to 2147483647. In testing, all array elements are the same [positive] integer (not particularly close to these integer limits). In my experience, the results surprise many students:

When all 10 array elements are 214748364, the average is 214748364 , as expected.

When all 10 array elements are 214748366, however, the average is -214748363 (at least on my Mac laptop).

In this case, each integer was roughly a factor of 10 lower than integer limits, and the sum encountered integer overflow with the second data set.

Altogether, such examples/exercises have the potential to highlight issues of integer overflow that can arise in practice when adding several integers.

3. Basic Properties of Floating-point Numbers and Casting

In upper-level courses, any students may understand the IEEE 32-bit and 64-bit standards for floating point numbers, and they may even have experience converting decimal numbers into their binary representations. Many [most?] students also understand how to cast from one data type to another—at least syntactically. However, students may not consider implications of representational error.

To illustrate, the following example is slightly modified from a warm-up exercise that has been given regularly by Prof. Tia Watts, Sonoma State University, at the start of a senior-level, programming languages course.[2]

EXAMPLE 3: CONVERTING DOLLARS AND CENTS (AS A REAL NUMBER) TO CENTS

Write a program that reads a number of dollars and cents as a real number (e.g., 10.24) and determines the most efficient way to make change (using common bills and coins).

For all but three (of about 25 students) in a recent class, a key part of their initial code submission (in this case in C++) utilized the code below (or quite similar):

While this approach works for many values, the result truncates the value (`amount` * 100), when converting to an integer. Thus, the amount 10.03 yields 1002 cents and 128.14 yields 12813 cents. Of course, for a reliably correct conversion, the amount (with representational error) must be rounded, not truncated.

For this class, most students had worked with binary representations and casting for several years, but apparently few realized basic consequences of representational error—at least when the

Example 3:

```
std::cout << "enter amount as a real number (dollars.cents): ";
std::cin >> amount;
int cents = amount * 100;

// subsequent processing, such as
std::cout << "amount: "  << amount
        << "; cents: " << cents << std::endl;
```

[2] This exercise originally was designed to review language syntax and establish mechanics for submitting assignments. For this course, students typically had taken at least one, and often two, courses in computer organization and architecture, which devoted a moderate amount of time to binary representation of numbers.

problem seemed to have a different focus. Prof. Watts, Sonoma State University, notes, "I don't know why it takes until [this upper-level course] for the importance of details like this to sink in - is it greater maturity? - or, just that this is the first time the issue has been worth points?"

Most students had worked with binary representations and casting for several years, but apparently few realized basic consequences of representational error—at least when the problem seemed to have a different focus.

4. Boolean Expressions and Loop Conditions with Floating-Point Numbers

At a number of schools, students have not encountered the potential for representational errors. For example, the decimal number 0.1 translates to a binary number with a repeating bit pattern that cannot be stored exactly in 32 or 64 bits.

With this in mind, the following examples/exercises can be particularly useful in attracting student attention..

EXAMPLE 4: AN INFINITE LOOP

Consider the C code to the right.

Due to representational error for the decimal number 1.0/10.0, the `val` variable never reaches exactly 1.0—just a value that is slightly too small or too big. Thus, the loop is infinite with either `float` or `double` data types. In my experience, students often express considerable surprise at this result, and running a program with this code can start class discussion of how to adjust Boolean expressions or loop structures to handle representational and computational error.

EXAMPLE 5: HOW MANY LOOP ITERATIONS?

The class discussion can continue with the simple change of the above code to

```
while (val <= end)
```

Here, the expected values in the loop for the `val` variable are 0.0, 0.1, 0.2, . . . 0.9, and 1.0, and `val` might be expected to terminate with the value 1.1.

When running this code when `val` is a `double` (using the 64-bit standard), the expected values are computed (subject to very small differences from representational/computational error). In contrast, when `val` is a `float` the loop stops one iteration earlier (when the value of `val` becomes slightly larger than 1.0). Initially, students may conclude that using `double` type rather than `float` can resolve such issues.

However, when the code is edited to go from 1000.0 to 1001.0 in increments of 0.1, students again may be surprised that the loop when `val` is a `float` enters the loop with value 1001.0 and the terminating value is roughly 1001.1, whereas when `val` is a `double`, the loop stops one iteration early. That is, the number of iterations is exactly reversed in these circumstances, and students discover that similar issues can arise in Boolean expressions for both types of variables.

Of course, if one wants to be certain that a loop iterates a specific number of times, then an integer control variable can yield the reliable result shown below.

Example 4:

```
float inc = 1.0/10.0; // float in C often uses
float val = 0.0;      // the 32-bit floating-point standard
float end = 1.0;

while (val != end){
   val += inc;
}
```

Example 5:

```
float val;      // using a double does not change the approach
int index = 0;  // int variable is adjusted
int end = 10;   //     by a factor of 10

while (index <= end){
   val = index / 10.0;
   index++;
}
```

Expanding potential troubles with a floating-point control variable, another illustrative loop divides the interval [1.0, 2.0] into n pieces, iterates through the endpoints of each piece, and counts the actual number of loop iterations:

```
xvalue = start;
width = (end - start) / n;
long counter = 0;
while (xvalue <= end) {
  counter++;
  xvalue += width;
  // if width too small to impact xvalue,
  //    stop after 10000 additional iterations
  if (counter == n + 10000)
    break;
}
```

```
Program to count number of iterations actually computed for division of [ 1.0,  2.0]
into n pieces using a while loop
                              Expected                      Actual Iterations
                n           Iterations             using floats           using doubles
              100                  101                      101                     100
             1000                 1001                     1000                    1001
            10000                10001                     9999                   10001
           100000               100001                    99865                  100000
          1000000              1000001                  1010000                 1000001
         10000000             10000001                 10010000                10000000
        100000000            100000001                100010000               100000001
       1000000000           1000000001               1000010000               999999918
      10000000000          10000000001              10000010000              9999999173
     100000000000         100000000001             100000010000             99999991726
    1000000000000        1000000000001            1000000010000            999911107321
```

In C or C++ on many machines, `xvalue` and `width` may be either floats (32-bits) or doubles (64-bits). Running the code yields the counts above.

For 32-bit floating-point storage (floats), the increment being added does not change xvalue once n reaches approximately 1000000, yielding an infinite loop without some type of check. In other cases, the output illustrates the actual number of iterations often varies from the desired number when a floating-point control variable is used.

Overall, such examples highlight some consequences of data representation for floating point control variables and can provide important guidance to students in writing both Boolean expressions and simple loops.

5. Error Accumulation During Addition of Floating-Point Numbers

Although students may [eventually] appreciate that representational and accumulated floating-point error can arise, they may not realize the important consequence that addition of floating-point is not guaranteed to be associative, and this can impact how programs should be written.

EXAMPLE 6: ARITHMETIC OF FLOATING-POINT NUMBERS NEED NOT BE ASSOCIATIVE

Of course, when only a fixed number of digits are stored, any additional digits cannot be stored directly, so numbers must be truncated or rounded to the digits available. Further, since this is a matter of basic storage, the same issue arises in both binary and decimal floating-point numbers. Thus, students need not have a detailed understanding of 32-bit or 64-bit floating-point representations to appreciate the difficulty.

For example, Figure 1 shows the computation of the decimal sum 1.000 + 0.0007 + 0.0007, where only four decimal digits can be stored. The specific computation depends upon where parentheses are inserted. Additional variation may arise, depending upon whether truncation or rounding is used to maintain 4-digits of storage.

In summary:

- Left associativity with truncating for (1.000 + 0.0007) + 0.0007 produces 1.000 as the arithmetic result.
- Left associativity with rounding for (1.000 + 0.0007) + 0.0007 produces 1.002 as the arithmetic result.
- Right associativity with truncating or rounding for 1.000 + (0.0007 + 0.0007) produces 1.001 as the arithmetic result.

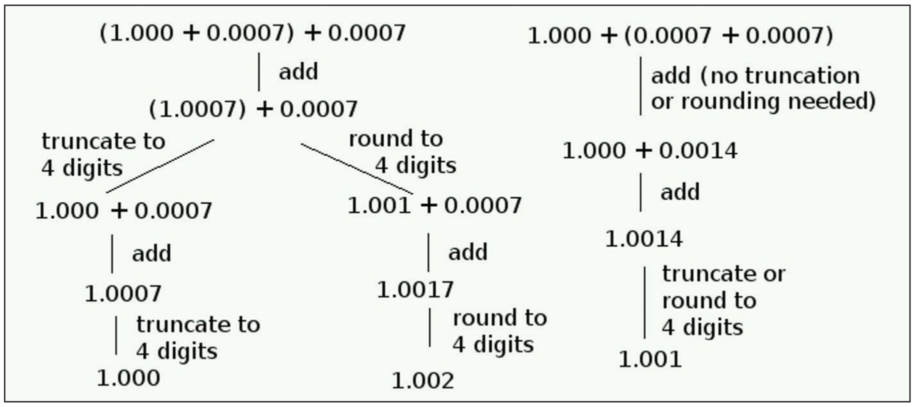

Figure 1: Computing1.000+0.0007 + 0.007 in two ways, storing 4 decimal digits.

In this example, regardless of whether truncation or rounding is used, left associativity produces a different result than right associativity. That is, addition within the computer is not associative, so many familiar properties of numbers no longer hold!

Also, this example illustrates an underlying observation that small numbers can get lost when added to much larger ones.

EXAMPLE 7: COMPUTATIONS IN LOOPS WITH MANY ITERATIONS

One way to illustrate accumulation of numerical error involves writing code that, in principle, should converge to a known value as the number of iterations increases.

For years, I used variations of the trapezoidal rule to approximate the area under a known curve (e.g., the area under $y = x^2$ on the interval [0, 4]). Since this function increases over the domain, computing trapezoidal areas from left to right adds large values to smaller ones, whereas computing right to left begins with relatively large functional values. Theoretically, as the number of trapezoids increases, the approximation should converge to the exact area. However, in practice, as the number of trapezoids increases, the approximations converges properly for a while, but eventually accumulated computational error increases and the approximations diverge. Further, adding smaller trapezoids first typically yields more accurate results.

Recently, I have experimented using another example: a series to approximate Pi. Specifically, consider the following series.

$$\pi = \sum_{i=0}^{\infty} \frac{(2i)(2i-2)(2i-4)\ldots(2)}{(2i+1)(2i-1)(2i-3)\ldots(1)}\left(\frac{1}{2}\right)^{i-1}$$

Here error analysis shows that when

evaluated for $0 \le i \le n$, the error is bounded by (1/2)n-1 [8].

Coding for this example provides several opportunities for helping students experience the consequences of floating-point processing.

* Code can initialize a floating-point variable to a seemingly large number of decimal places, but the actual value stored may not reflect the accuracy of the initial value specified. For example, consider the initialization (in C) and subsequent printing

```
double actualPi = 3.14159265358979323846264433;
printf (" %23.30lf", actualPi);
```

The actual value printed is 3.14159265358979311600, which is accurate to 15 decimal places, but accuracy beyond that point is lost—a double can only store about 16 digits reliably (in this case the initial 3 plus 15 digits after the decimal point). Digits beyond those places may be printed, but they may not be accurate.

- When the series is computed for $0 \le i \le n$, a simple approach is to store the n+1 terms in an array and then add the results with i going from 0 to n and again from n to 0. (A full C-based program is given in [9].)

Experimenting with this approach can be revealing for students. Results run on the author's MacBook Pro yield the following selected output below, in which results are printed to 16 decimal places.[3]

Reviewing this output, we note the following:

- adding smaller terms first (i.e., adding ending with i = 0) consistently yields less accurate results than adding smaller terms last;
- after about n = 50, adding the smaller terms first yields a correct approximation to Pi to 16 decimal places; and
- adding larger numbers first improves as n increases until about n = 50. Thereafter, the additional terms are too small to impact the overall computation, and the approximation continues with the same error.

For readers interested in pursuing examples and/or exercises on consequences of numeric representation, [11] provides a reading for students, and [9] a corresponding lab.

6. Conclusion

From the beginning of the ACM recommendations for undergraduate computer science in 1968 [1, pp. 156, 170], the topic of the binary representation of numbers (both integers and floating-point formats) has been included in curricular guidelines. Often, course work may emphasize specific formats, and students learn how to convert between decimal and binary numbers. This work certainly provides worthwhile background that can relate to circuit design and explain processing behaviors.

However, conversations with many computing faculty suggest that coverage of numeric formats may not extend to practical consequences for writing programs. Further, the practices and examples given here suggest that students need not have a deep and detailed understanding of underlying numeric formats to appreciate potential issues and apply worthwhile programming approaches. Details of numeric formats can explain specific behaviors, but high-level principles may be sufficient to help guide students in writing programs.

Altogether, consequences of the binary representation of data can have a substantial impact on the writing of programs, and

Consequences of the binary representation of data can have a substantial impact on the writing of programs, and this seems to be an important message for students to experience—independent of whether they have mastered details of specific formats.

```
                         approximation     actual value (to 16 places)     error

number terms:   21
 i goes 0 to 20: 3.1415919276751456    3.1415926535897931    0.0000007259146475
 i goes 20 to 0: 3.1415919276751469    3.1415926535897931    0.0000007259146462

number terms:   41
 i goes 0 to 40: 3.1415926535892931    3.1415926535897931    0.0000000000005000
 i goes 40 to 0: 3.1415926535892940    3.1415926535897931    0.0000000000004992

number terms:   51
 i goes 0 to 50: 3.1415926535897922    3.1415926535897931    0.0000000000000009
 i goes 50 to 0: 3.1415926535897927    3.1415926535897931    0.0000000000000004

number terms:   61
 i goes 0 to 60: 3.1415926535897922    3.1415926535897931    0.0000000000000009
 i goes 60 to 0: 3.1415926535897931    3.1415926535897931    0.0000000000000000
```

[3] Although the specific results presented here were generated by a C program running compiled with gcc on a MacBook Pro, similar results have been obtained with other compilers, with C++ and Java, and on Windows and Linux platforms.

this seems to be an important message for students to experience—independent of whether they have mastered details of specific formats.

References

1. ACM Curriculum Committee on Computer Science, *Curriculum 68*, Communications of the ACM, Vol. 11, No. 8, March, 1968; https://dl.acm.org/doi/10.1145/362929.362976; accessed 2023 Aug 27.
2. ACM/IEEE-CS Joint Task Force on Computing Curricula, *Computing Curricula 2001 Computer Science*, December 15, 2001, URL: https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2001.pdf; accessed 2023 Aug 27.
3. ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013*. ACM Press and IEEE Computer Society Press, December 2013.
4. ACM/IEEE-CS/AAAAI Joint Task Force on Computing Curricula, Computer Science Curricula 2023, Version Gamma, August 2023; https://csed.acm.org/wp-content/uploads/2023/09/Version-Gamma.pdf; accessed 2023 Sep 20.
5. Goldberg, D., What Every Computer Scientist Should Know About Floating-Point Arithmetic, posted by Oracle Corporation, an edited reprint of an article in Computing Surveys, March, 1991; https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html; accessed 2023 Sep18.
6. Hennessy, J. L., and Patterson, D. A., *Computer Organization* . . . (The Morgan Kaufman Series in Computer Architecture and Design) This series includes several editions, such as Computer Architecture and Design Risc-V (2020) and Computer Architecture: A Quantitative Approach (2017). Specific comments here related to the Risc-V (2020) version.
7. Overton, M. L., *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM (Society for Industrial and Applied Mathematics, 2001; https://cosweb1.fau.edu/-jmirelesjames/ODE_course/Numerical_Computing_with_IEEE_Floating_Point_Arithmetic.pdf; accessed 2023 Sep17.
8. StackExchange, Series that converge to $\prod$ quickly, https://math.stackexchange.com/questions/14113/series-that-converge-to-pi-quickly; accessed 2023 August 25.
9. Walker, H. M. Lab: "Consequences of Data Representation on Programming", https://blue.cs.sonoma.edu/-hwalker/courses/415-sonoma.sp24/readings/data-rep-consequences.php; accessed 2023 Oct 20.
10. Walker, H. M., C-program, pi-approx.c, https://blue.cs.sonoma.edu/-hwalker/courses/415-sonoma.sp24/assignments/pi-approx.c; accessed 2023 Aug 25.
11. Walker, H. M. Reading: "Consequences of Data Representation on Programming", URL: https://blue.cs.sonoma.edu/-hwalker/courses/415-sonoma.sp24/readings/data-rep-consequences.php; accessed 2023 Aug 25.
12. Walker, H. M., "What is the Payoff?", *ACM Inroads*, Vol. 13, Issue 1, March 2022, pp. 10-14.

Henry M. Walker
Department of Computer Science
Grinnell College
Grinnell, Iowa 50112 USA
walker@cs.grinnell.edu

DOI: 10.1145/3627171

by David P. Bunde, *Knox College*, Zack Butler, *Rochester Institute of Technology*, Christopher L. Hovey, *University of Colorado Boulder*, and Cynthia Taylor, *Oberlin College*

Conversation with a Prominent Propagator: Carl Haynes-Magyar

Encouraging faculty to adopt new, high-impact teaching practices, tools, and curriculum in Computer Science (CS) undergraduate education requires intentional planning and sustained effort. This article is the next installment in the series of interviews with prominent propagators: members of the CS education community who have successfully spread pedagogical or curricular innovations [2–4]. The goal is to capture knowledge and experiences that others can use to propagate their own teaching projects.

In this article, we interviewed Carl Haynes-Magyar, Presidential Postdoctoral Fellow at Carnegie Mellon University's School of Computer Science in the Human-Computer Interaction Institute. Carl is the creator of Codespec [5], an online programming tool, which is designed for self-guided programming practice in a variety of modalities, including scaffolded programming tasks such as block-based languages and Parsons Problems, as well as free-form coding. Codespec is built on IDEAS+: inclusion, diversity, equity, accessibility, sexual orientation and gender identity awareness, and justice [15]. Codespec scaffolds learners' programming skills development through a range of research-based problem types and learning support features. Carl also conducts research on tools and processes that support learners' development of self-regulated learning skills [10–13].

Below are highlights of the interview, which ran approximately an hour and a half.

Carl Haynes-Magyar

The transcript has been edited for clarity and style, and our interviewee has generously provided citations to relevant work.

Q: DESIGNING FOR A DIVERSITY OF USERS AND CREATING A SENSE OF BELONGING SEEMS TO BE CENTRAL TO CODESPEC. CAN YOU PLEASE TELL US A LITTLE ABOUT DESIGNING FOR A SENSE OF BELONGING AND HOW THAT EXPRESSES ITSELF IN YOUR WORK?

CHM: Coincidentally enough, I was just talking to some trans and non-binary (TNB) computing education researchers about a workshop they held that investigated the intersectional identities of TNB learners as they flow through the ecosystem of computing [19]. LGTBQIA2S+[1] learners are more likely to think about leaving computing because of a low sense of belonging [21]. And yesterday, I talked to a group who was awarded an NSF grant and were interviewing different stake-

[1] This acronym broadly refers to diverse gender-, sex-, and sexuality-based identities including Lesbian, Gay, Transgender, Bisexual, Queer, Questioning, Intersex, Asexual, Aromantic, Two-Spirit, Gender Fluid, etc.