Development and illustration of a framework for computational thinking practices in introductory physics

Daniel P. Weller,[1,2] Theodore E. Bott,[1] Marcos D. Caballero,[1,3,4] and Paul W. Irving[1]

[1]*Department of Physics and Astronomy, Michigan State University, East Lansing, Michigan 48824, USA*
[2]*School of Mathematical and Physical Sciences, University of New England, Biddeford, Maine 04005, USA*
[3]*Department of Computational Mathematics, Science, and Engineering and CREATE for STEM Institute, Michigan State University, East Lansing, Michigan 48824, USA*
[4]*Department of Physics and Center for Computing in Science Education, University of Oslo, Oslo 0316, Norway*

(Received 15 May 2021; revised 4 February 2022; accepted 8 June 2022; published 25 July 2022)

Physics classes with computation integrated into the curriculum are a fitting setting for investigating computational thinking. In this paper, we present a framework for exploring this topic in introductory physics courses. The framework, which was developed by reviewing relevant literature and acquiring video data from high school classrooms, comprises 14 practices that students could engage in when working with Glowscript VPython activities. For every practice, we provide in-class video data to exemplify the practice. In doing this work, we hope to provide ways for teachers to assess their students' development of computational thinking and give physics education researchers a foundation to study the topic in greater depth.

DOI: 10.1103/PhysRevPhysEducRes.18.020106

I. INTRODUCTION

Computation has transformed from being a tool that assisted scientific research to being fundamental to the very meaning of doing science. Computational thinking (CT) is an important emerging set of skills for students in the 21st century [1]. It is the underlying set of practices that scaffold the "doing" of computation. This topic was first promulgated in an influential 2006 article by Jeanette Wing who described it as "thinking like a computer scientist" [2]. This seminal work expounded the idea that computer science education should be thought of as more than just programming; it is a conceptualization of the fundamental skills required to solve complex problems. More recently, CT has been defined as "the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer —human or machine—can effectively carry out" [3]. Recently, global education efforts have focused on providing students experience with CT [4–6]. The prevalence of computational thinking in modern educational and professional spaces motivates our interest in this subject.

Although computational thinking was initially proposed by the computer science community, it has since been incorporated into disciplinary science education standards [7,8]. For example, the Next Generation Science Standards (NGSS), which have been adopted by a majority of states in the United States, emphasize using mathematics and computational thinking as one of the eight major scientific practices that all students should encounter in their K–12 education [9]. For example, in grades 9–12, an example of CT is when students "create/revise a computational model or simulation of a phenomenon." This is a broad expectation that could be applied to multiple disciplines. However, expectations have also been formalized for specific disciplines. For instance, in the case of high school physical science, one performance expectation states, "Students who demonstrate understanding can create a computational model to calculate the change in the energy of one component in a system when the change in energy of the other component(s) and energy flows in and out of the system are known" [9]. This benchmark provides one example of how CT could manifest in high school physics classrooms. The increased expectancy around CT learning outcomes indicates a need to study it from a physics education research (PER) perspective.

Computationally integrated physics courses are a rich area for exploring CT. The repository of exercise sets from the Partnership for Integration of Computation in Undergraduate Physics features a plethora of computational activities incorporating CT practices [10]. Additionally, the 2020 conference report about computational thinking from the AAPT headquarters [11]. Physics courses that utilize computational modeling activities are often termed "integrated" courses because they do not focus on programming as the primary objective. Computationally integrated courses infuse computational ideas within existing physics course content.

Preliminary work has been carried out to study integrating computation within physics classes in primary school, secondary school, and higher education levels. A theoretical framework was proposed by Sengupta *et al.* in 2013, which described a modeling cycle with CT ideas underlying the entire sequence [12]. Their CT-infused modeling cycle includes steps like scientific inquiry (i.e., developing understanding of scientific phenomena and modeling practices), algorithm design (i.e., developing understanding of programming techniques), and engineering (i.e., iteratively refining a model). This cycle is underscored by CT because it is iterative and involves developing computational procedures to represent physical models. More recently, Orban and Teeling-Smith discussed the manifestation of CT in introductory physics classes [13]. Their work, as well as that of Petter-Sand *et al.* [14], highlight the connection between sense-making and computation in the physics classroom. Computational modeling activities are apposite settings for investigating CT at the introductory physics level.

There is still a lack of agreement around assessment strategies for CT in introductory physics [15–17]. Some work has been done by Swanson and colleagues to develop an assessment framework for CT in secondary science classrooms [18]. However, in an attempt to be broad and all encompassing, this framework omits the nuanced details required to explore CT in-depth for each specific discipline, such as physics versus chemistry or biology. We claim that the universality of one CT framework is problematic when applying it to a different context, as every classroom is unique. A primary goal of this manuscript is to develop a framework specifically aimed at investigating CT at the introductory physics level. We are interested in what the students are doing in the classroom (i.e., practices) so that research questions around activity design and integration can be answered. At this stage in CT research, it is useful to understand how design decisions around activities can influence the CT practices that occur, rather than simply assessing students' development through a pre-post test approach.

Herein, we lay the foundation for a computational thinking framework specifically aimed at physics educators and researchers interested in teaching and studying CT when employing computational activities at the introductory physics level. For researchers, we give a detailed description of the practices in our context and discuss some preliminary results from its initial implementation. For physics educators, we aim to vivify what CT practices can look like in the classroom to guide future activity design and assessment. We present the following research question to elaborate on our goals:

- How do computational thinking practices manifest in introductory physics classrooms?

Answering this research question involves blending two perspectives. The first being a theoretical literature-based one, the other being an empirical evidence-based one. This report will review previous literature to develop a framework specific to our context, and then we will provide video data from in-class interactions between students to serve as evidence for these practices. Ultimately, we hope to highlight the importance of context (i.e., discipline, class level, pedagogical strategies, computational platform, etc.), while also providing an actionable set of CT practices in computationally integrated physics classrooms.

II. FRAMEWORK DESIGN AND RESEARCH METHODOLOGY

A. Framework development timeline

The computational thinking framework proposed in this report resulted from combination of reviewing previous

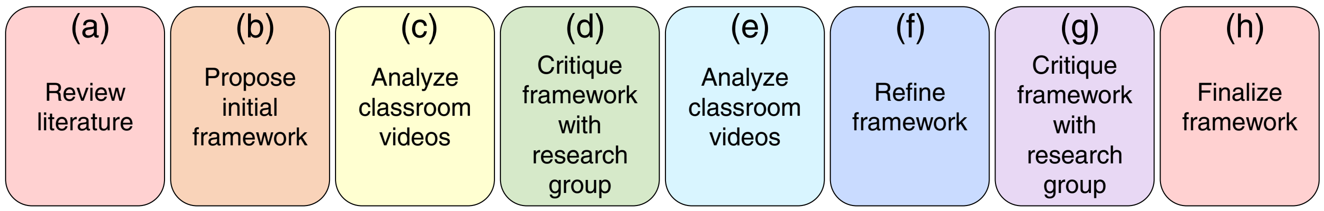

FIG. 1. Timeline of the methodological steps that were taken to develop the CT framework proposed in this report. (a) First, a literature review was conducted to identify works pertaining to computational thinking practices. (b) An initial list of practices was proposed with 17 uncategorized, standalone practices. (c) The initial practices were used to analyze in-class video data and find examples of the practices from students working with computationally integrated physics activities. (d) An initial framework was constructed and shared with external researchers to provide feedback on how the framework could be improved in terms of its relevance and precision of definitions. (e) Classroom videos were analyzed again to find examples that served as evidence for the updated practices. (f) The framework was refined based on findings from in-class data and feedback from researchers. (g) The framework was critiqued by a PER group to discuss its relevance to physics, as well as any confusing language in the framework's definitions. (h) The finalized framework comprised 14 practices in 6 categories.

literature and analyzing of in-class video data. Figure 1 displays a timeline of the development process for our CT framework. The process began with a review of the relevant literature [Fig. 1(a)]. Although there are innumerable articles that discuss computational thinking, we limited our selection of articles to only include primary research articles that proposed CT practices and descriptors of such practices. The inclusion criteria for articles included mention of practices, or the actions that students take when engaging in CT. Some relevant search terms (combined interchangeably) were computational thinking, computation, modeling, physics, K–12, programming, debugging, coding, data, and problem-solving. In total, 32 scholarly articles were initially reviewed. Then, we narrowed our review to closely examine works that were frequently referenced among the literature. Works that did not have clear categorization schemes were excluded, and we focused on studies that presented theoretical frameworks, rather than empirical works. To better match our specific context, we narrowed the articles to ones that only focused on K–12 education rather than higher education. This reduction resulted in the selected works discussed in Sec. II B. After choosing these prominent CT studies as a basis for our initial framework, we read the papers and identified major similarities and differences among the papers. This process helped to comprehend the critical aspects that we should look for then informing our analysis process.

We decided to develop a framework for our specific context [see Fig. 1(b)]. The goal of this development was to encompass ideas that were highlighted in previous frameworks while also sorting those ideas through the filters of physics and high school classrooms. The intention was to use the practices from previous frameworks as an initial analysis tool, with the assumption that we would have to iterate, edit, ignore, discover, and redefine the practices to be relevant to introductory physics. At first, we were trying to apply some of the previous frameworks directly to our context, but in practice, this proved to be more difficult than anticipated. We found that many broad CT frameworks were not able to capture the nuanced details of what we were observing in video data or what was being discussed by teachers in informal interviews [19]. At this stage, our goal was to be as inclusive as possible, resulting in a large list of initial practices. Our initial list consisted of 17 practices, and all practices were standalone (i.e., no categorization scheme was used to combine related practices).

The step in Fig. 1(c) was our first attempt at applying the practices to in-class data. We watched 12 h of in-class video data from 2 different classrooms (details of video analysis techniques will be provided in Sec. II C). Our goal was to notice which practices from the initial list were appearing as we expected to see them, while also looking for interesting moments that were not encapsulated within our current set of practices. For example, systems thinking, which is an entire category in other prominent CT frameworks [20], was not coded at all in the data that we analyzed. Similarly, modularity and parallelization were difficult to find evidence for as well. After reviewing these practices with our team, we found that some were not appearing in our context because they were either too complex, too broad, or overly specific to our programming environment (Glowscript VPython). On the other hand, we noticed many striking examples related to student affect (e.g., positive attitudes, or moments of frustration), which were not covered by our initial framework. These findings provided evidence that supported the inclusion or exclusion of practices in the future framework iterations.

The initial version of the framework was critiqued by a group of computer education researchers [see Fig. 1(d)]. We initially provided descriptions of practices with video examples for discussion. Their critique helped us realize that our CT practices should be focused on moments when students were working with computers or programming, so as to not conflate CT practices with general problem solving practices. Some practices seemed to be too broad to be considered exclusively related to CT (e.g., planning, iterative problem solving). They suggested that we eliminate some practices that were too complex or high level for introductory physics students (e.g., systems thinking, modularity, and parallelization). It was recommended that we use categories to link related practices together. Instead, these practices might simply be considered general problem-solving practices. By the end of this process, 3 practices were removed (planning, systems thinking, and parallelization), the definitions of several practices were refined, and related practices were combined into categories. After this critique, 14 practices existed within 8 categories.

Subsequently, another round of video analysis was conducted to identify instances of CT in our context with the updated framework [Fig. 1(e)]. One interesting amendment was our refining of the descriptor for the debugging practice. In our initial video analysis, we only coded a practice as debugging if the students were working through a fatal error that caused the program to not execute the code entirely. However, in our second round of video analysis, we found that students were often spending a lot of time just trying to make the code do what they wanted it to do, even though the code did not necessarily exhibit a fatal error. This finding supported our decision to redefine debugging as students working through fatal errors or “unexpected behaviors.” The language of unexpected behaviors for debugging is consistent with some literature sources, and numerous instances of video evidence were found to support this redefinition. An updated framework resulted from this second pass at the video data [Fig. 1(f)]. The updated framework still contained 14 practices, but in 6 different categories, and the practice descriptors were updated to more closely match the experience of introductory physics students.

The refined framework was presented for another round of critiquing to gain further insight about it application for physics education researchers [Fig. 1(g)]. Many of these individuals had experience integrating computation into their own classrooms. We learned that the usability of our framework could be improved by concisely defining practices and providing a wealth of examples to explain the variation of CT practices. The researchers noted that data practices did not make sense in its current state. Before this critique, data practices were there own CT practice and it was defined as “the ability to manipulate and analyze data computationally through reducing, fitting, filtering, averaging, organizing, or computing uncertainties.” Feedback in this session described how this was too broad of definition to find many meaningful examples of data practices in the selection of examples that we provided. Instead, it was proposed that data practices should be their own category of practices, with more specific actions being contained within that category. At this point, we also decided that a category for affect-based practices should be created. This suggestion led to our development of the practice of demonstrating affective dispositions towards computation.

Finally, following multiple critiques and iterations of video analysis, the finalized framework contained in this report was proposed. The finalized framework consists of 14 practices in 6 different categories. The categories encompass ideas related to thinking about computation (i.e., extracting physical insight), working with computation (i.e., building computational models), and using computation to make claims or draw conclusions about a model (i.e., data practices). The framework also ended up containing more broad practices like debugging, group work, and affect-based ideas. The complete framework will be discussed in Sec. III.

B. Review of computational thinking literature employed in framework development

Due to the level of physics classroom we had access to for this study, our literature review and presentation of in-class evidence mostly revolves around the context of introductory physics. An important point of emphasis is that the literature included here is not meant to fully encompass all research pertaining to CT. It is not our intention to provide a comprehensive overview of previous research on integrating computation into the discipline of physics. Rather, we seek to present a contextually diverse subset of CT frameworks. Our approach allows for the emergence and transformation of CT skills from other dissimilar contexts to our particular case.

Barr and Stephenson provide a commentary on integrating CT into K–12 education [21]. Difficulty arises because there is no widely agreed upon definition of computational thinking, which is discussed by nearly every framework reviewed herein. The items in their framework are called core computational thinking concepts and capabilities. Many of these concepts or capabilities are evident in other frameworks, namely, problem decomposition, abstraction, algorithms and procedures, and simulation. Other concepts or capabilities, such as automation, are less common in other frameworks.

Berland and Lee explored how computational thinking manifests when playing strategy-based board games [22]. The framework in this study is better described as a coding mechanism, which is applied when observing student groups playing the game together. The framework consists of five core aspects of computational thinking. Each category is presented in a table that includes descriptions, rationale behind inclusion, and examples of student dialogue that was coded in each respective category.

Brennan and Resnick’s study focuses on CT in the Scratch programming environment [23]. The framework is organized along three dimensions: concepts, practices, and perspectives. Their seven concepts resonate specifically with programming environments. Their computational practices section is heavily focused on students learning to work with code. The computational perspectives category focuses on students expressing themselves creatively through computation, enriching computational experiences through interaction with others, and questioning the complicated nature of technology.

The framework proposed by Weintrop and colleagues is a four-category taxonomy of computational thinking practices [20]. The categories include data practices, modeling and simulating practices, computational problem-solving practices, and systems thinking practices. Each category contains five to seven CT practices. It is worth noting that besides the set of recommendations presented in the next article from AAPT, the Weintrop framework has been utilized more often than other for the context of physics [13,24]. Weintrop *et al.*’s work involved the analysis of 12 physics lesson plans and the perspectives of professional physicists. This study successfully contextualized CT to math and science contexts. In essence, our work builds upon the work of Weintrop *et al.* but further contextualizes CT practices to high school physics classrooms using VPython.

The recommendations for computational physics from the American Association of Physics Teachers (AAPT) was written to increase emphasis on computation within introductory physics [25]. The paper establishes a set of skills, which are organized into either technical computing skills or computational physics skills. There are only three technical computing skills: processing data, representing data visually, and preparing documents and presentations that are authentic to the discipline. As for the computational physics skills, some of them are significantly different from those included in other frameworks, such as “translating a model into code” and “choosing scales and units.” The majority of the other skills, including “subdividing a model into a set of computational tasks” (i.e., decomposing) and “debugging, testing, and validating code” (i.e., debugging) matched well with existing CT literature [25].

Shute *et al.* aimed at examining literature around CT to resolve conclusively the lack-of-definition issue that has plagued CT researchers since its conception [26]. They claim the four most common components of CT from the literature are abstraction, decomposition, algorithms, and debugging. The authors used these four practices as the basis for their framework structure. Other practices that are included in the framework are either standalone practices alongside these four (e.g., iteration and generalization) or they can be considered one element of the primary four (e.g., modeling is a subcomponent under abstraction).

Lyon and Magana that took a similar approach to building their CT framework [27]. A key point made in this work is that the specific context was chosen ("a required upper-division senior engineering capstone course on food and pharmaceutical processing within a biological engineering department" [27]) for because (1) most students in the course had limited computational experience, (2) there was a more even distribution of males and females, and (3) the research team was more closely involved with the course content. These justifications highlight the authors' belief that CT could manifest differently in courses that have varying computational experience or an uneven gender distribution amongst participants. Their practices consisted of abstraction, algorithmic thinking, evaluation, generalization, and decomposition. They did not, however, take these definitions verbatim from the work, but rather refined them through thematic analysis and reflection about the coding rubric in a similar process described in this paper.

Of the studies reviewed, the work of Palts and Pedaste resonates most with our work [28]. Their study is based on a comprehensive literature review of previous CT studies, which were then filtered through a lens of CT problem solving. This results in ten CT skills within three categories: defining the problem, solving the problem, and analyzing that solution. It includes several themes that also occur in our framework (albeit with subtle naming distinctions), such as decomposition, data collection or analysis, algorithmic design, and generalization. One notable deviation from other frameworks is the lack of affect-based practices.

We would like to highlight the fact that context is an influential factor on how CT manifests in all these studies. Whereas there is a generalizability to most previous CT frameworks, we are interested in the pieces that are applicable to our specific context. Most of the frameworks exhibit different structures, groupings, and grain sizes. Whereas some frameworks utilize categorizations schemes, others present CT as standalone practices. In our framework, we grouped and tiered some but not all practices. Grouping was based on patterns observed in our video analysis. Moreover, many of the frameworks consider CT on the scale of ideas, or actions, or dispositions. For our work, we mostly focus on students' behaviors or actions that elicit CT so that practitioners can more clearly identify CT with actionable examples. Lastly, affect-based ideas are especially relevant to activities where students are working in groups or working through a particularly challenging task (i.e., coding).

C. Research context, data collection, and video analysis details

The context of this work is computationally integrated high school physics classes in Michigan. The instructors involved in our study participated in a professional development series called Integrating Computation in Science Across Michigan (ICSAM). This program is an NSF-funded project where teachers learn to program and teach computational modeling to their students. In ICSAM, the curricular design is solely at the teacher's discretion. There is no set curriculum that the workshop intends to establish in every participant's classroom. Rather, each instructor decides how computation will be integrated in their respective environment.

The programming language that teachers use for all computational modeling activities is Glowscript VPython [29]. This language is particularly well designed for modeling objects and events in a 3D environment, as it was specifically designed for use in physics classrooms. The ICSAM model makes use of "minimally working programs" (MWPs) to simplify the more difficult aspects of coding, to provide scaffolding for students, and to keep the focus on physics even during computational activities [30]. Minimally working programs are beneficial because students can start activities by interacting with the physics immediately, while not being slowed down with creating a computer program from scratch.

Overall, this research report examines 6 different computational activities from 2 different classrooms. The project as a whole gathered data from 15 teachers' classrooms totaling about 170 h of in-class audio or video data. We focused on the activities from Michael's and Liam's (pseudonyms) classrooms, and we chose activities where the videos featured a large amount of interactions between group members. Michael's class was an AP physics (third and fourth year students), and we looked at his projectile motion, river crossing, and spring energy activities. Liam's class was Physical Science 2 (first and second year students), and we analyzed his colliding crates, momentum conservation, and block on a ramp activities. See the appendices A and B for details about both classrooms and the activities analyzed.

Audio and video data were gathered from high school classrooms around the state of Michigan. Videos were acquired by researchers visiting the classrooms of teachers participating in our professional development series. To gather data, a camera on a tripod was positioned to record students' discussions, body language, and equipment.

A microphone was placed in the middle of student groups to record their conversations.

We coded video examples and conducted a thematic analysis of in-class data using MAXQDA [31–33]. A mixed *a priori* (i.e., predetermined) and *a posteri* (i.e., generative) coding scheme was employed to analyze videos [34]. The initial phase of our video analysis involved watching the data and coding students' behavior. For example, one behavior that frequently occurred involved students deciding to focus their attention on one small computational task. At first, behaviors like breaking the code into segments and deciding which segment to focus on were grouped together because they fit a theme of trying to comprehend (i.e., gain insight about) the task at hand. The breaking down of code into sections (later to be termed decomposing) was determined to be distinct from deciding on which of those sections was the most important to focus on (later to be termed highlighting and foregrounding). Once these two codes had been identified, we returned to the filtered framework and examined it for alignment. If alignment was found, then the emergent theme would be paired with the practice from our initial list of practices. If no alignment was found, then the theme was written up as an emergent CT practice.

In regards to the coding of individual CT practices in the video data, we used behavioral markers to indicate the beginning and end of practices. Markers that often indicated the start or end of a CT practice were when students moved on to the next portion of their assignment, when students had a long period of silence in their discussion (i.e., 10–15 sec or more of not talking), or when students shifted their attention to a teacher or different group member. A complication with the individual identification of practices was that it was possible for some CT practices to co-occur with each other, especially in the case of debugging. For example, students would commonly employ other practices like utilizing generalization (e.g., if it worked previously, the same resolution should work here) or decomposing (e.g., breaking down different parts of code to more precisely understand why an error is occurring) while also debugging. We decided that co-occurrences would be coded as such rather than trying to discern distinguishable cutoffs for separate practices.

The thematic coding process was conducted by two researchers, and themes went through a peer review and negotiation process before being finalized. Once the pair of researchers had constructed a framework from codes that emerged from the data, it was tested for validity by giving it to a third researcher. The third researcher coded the same dataset, and the framework underwent a refinement process based on this review. Refinement included making definitions more concise, providing examples in the codebook for each different practice, and adding common markers for the beginnings or ends of a practice. In the end, separate raters came to agreement on more than 85% of the video segments provided (approximately 1 h of the total 12 h of data analyzed).

III. RESULTS: COMPUTATIONAL THINKING PRACTICES IN INTRODUCTORY PHYSICS

Our computational thinking framework, shown in Fig. 2, contains 14 practices within 6 different categories. When working on a specific computational task, students engage in the first three categories of CT practices: extracting computational insight, building computational models, and data practices. The other three practices (i.e., debugging, working in groups on computational models, and demonstrating affective dispositions with computation) exist as their own categories of independent CT practices. We will first discuss details about the categorization scheme before discussing each practice.

Extracting computational insight means perceiving and identifying the essential components of a computational model. It involves viewing the model as an abstraction of a real-world physical phenomena. Abstracting is included as top-tier CT practice in a vast number of frameworks, and

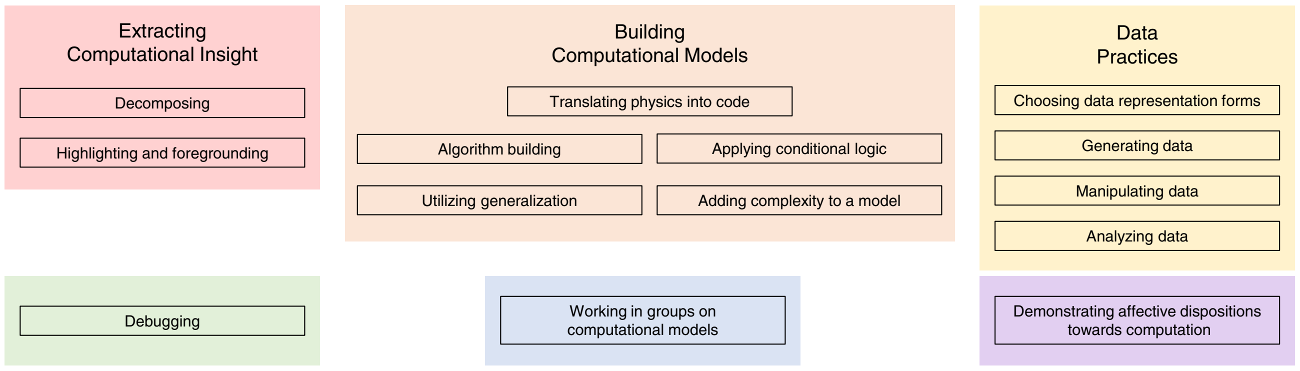
Extracting Computational Insight
Decomposing
Highlighting and foregrounding
Building Computational Models
Translating physics into code
Algorithm building
Applying conditional logic
Utilizing generalization
Adding complexity to a model
Data Practices
Choosing data representation forms
Generating data
Manipulating data
Analyzing data
Debugging
Working in groups on computational models
Demonstrating affective dispositions towards computation

FIG. 2. Outline of the practices in our computational thinking framework. The framework has 14 practices comprising 6 distinct categories. The categories of extracting computational insight, building computational models, and data practices all contain several practices within them. The categories of debugging, demonstrating dispositions towards computation, and working in groups on computational models exist as standalone practices.

this category aims to address the large grain version of abstraction. There are three practices within extracting computational insight: decomposing, modular thinking, and highlighting and foregrounding.

We define building computational models as using a computer to create an abstract representation of a physical system or phenomenon. For teachers administering computationally integrated physics problems, computer simulations will be the main modeling tool for students. An important note is that this category is associated with building computational models, and as such, all of the practices contained in this category pertain to students designing, enacting, and modifying their models. There are five practices contained within building computational models: translating physics into code, algorithm building, applying conditional logic, utilizing generalization, and adding complexity to a model.

Our category of data practices attempts to encapsulate the many different ways that students can gather information from computation. This includes creating or collecting data, preparing data for analysis, making claims from data, and producing data visualizations. Data practices as a whole are discussed extensively in most of the references included in our review. It is important to note the way that Glowscript VPython is used limits the ability to produce and store data. This is because students do not use arrays to store data, as might be typical in other coding environments. Instead, variables store data at a specific instance of the computational model, and much of the data is in the form of a visual output (rather than numerical values).

The categories of debugging, working in groups on computational models, and demonstrating affective dispositions towards computation are each in their own category containing a single practice. Generally, debugging occurs in many different forms, so it does not fit cleanly within any other category. Working in groups on computational models and demonstrating affective dispositions relate to student attitudes and interpersonal skills, both of which seem to be exhibited on a different grain size than all other practices in the framework. These last two practices are much more macroscopic, and thus, we have assigned them to their own categories.

A. Decomposing—Separating a computational problem into a series of manageable tasks

Numerous frameworks highlight the practice of breaking down large problems, complex systems, or multifaceted computational models into smaller pieces. The development of a computational model can be a multistep process and might incorporate many different concepts at one time. As a result, decomposition is one of the most universal practices included in CT frameworks. Most notably, Shute *et al.* emphasize the connectedness of the decomposed pieces that make up an entire model or solution [26]. Aside from minor language distinctions, each framework appears

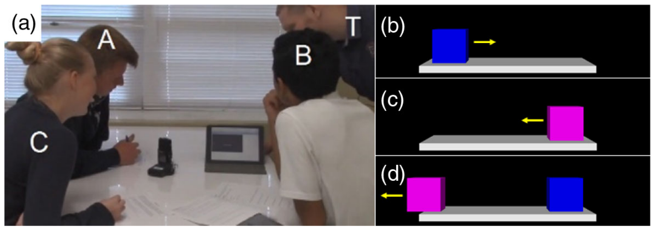

FIG. 3. (a) A group of three students (pseudonyms A, B, and C) discuss their computational model with the teacher (T). The students' model displays (b) first a blue box moving to the right, (c) then the blue box stopping on the right side of the platform and a pink box appearing after the blue box completes its motion, and (d) lastly the pink box continuing its motion infinitely to the left of the platform. Yellow arrows in (b), (c), and (d) have been added to illustrate the direction of motion for each crate at different instances.

to generally agree on the meaning of decomposing. Needless to say, the general notion of reducing complex systems or problems to their simpler components is well received as a core element of CT.

In the following episode, the students demonstrate decomposing while troubleshooting some issues in their model of colliding crates. They are trying to make the two crates move toward each other simultaneously such that they stop moving when they reach each other in the center (see Fig. 3). Ultimately, this program will be used to model an elastic collision between the two objects, but first, the students are simply trying to get both cubes to move toward each other at the same time and stop when they meet. When the students run their program, it initially displays a blue crate moving to the right [Fig. 3(b)]. Then, when the blue crate reaches the right side of the platform, it stops its motion, and a pink crate appears at the position where the blue crate stopped [Fig. 3(c)]. Subsequently, the pink crate moves to the left and continues its motion infinitely, even after it extends beyond the platform object [Fig. 3(d)]. The students call their teacher over for guidance.

B: So here's where we are. [Student B runs the program to show the teacher the output of their program.]

T: You have box 1 that flies across, and then box 2 comes into play, and flies across.

A: And, off… [The animation shows the second (pink) box moving infinitely to the left beyond the platform.]

B: And then it doesn't stop.

T: No… Oh, it doesn't stop!

B: Nope, we don't really know why. So, there's some problems. So, there's a delay, that's problem number 1. And, that is another problem. [Student B points at the second (pink) box still moving farther off-screen.]

T: Yeah, yeah, so which problem do you want to fix first?

B: The time delay.

This is an example of decomposing problems because the group is dividing one overall task (i.e., making both crates appear at the same time and simultaneously move toward each other) into two smaller tasks (i.e., fixing the time delay before the pink crate appears, and stopping the motion of the pink crate as it moves off the platform). In this case, the students identify multiple issues with their simulation. They choose to address the time delay first, and this serves as a chance for the group to re-focus their efforts toward a more manageable task. While the example provided here occurs with the teacher being present, student B actually decomposes the problem without any specific prompting from his instructor. This example provides some evidence that the teacher's presence could lead to students more easily engaging in the practice of decomposing. It should be noted that decomposing does not always occur in the context of troubleshooting or debugging. Another easily conceivable example could emerge when students are first interpreting the different lines of code in an MWP.

B. Highlighting and foregrounding—Perceiving the most important features of a computational task to enhance understanding, focus on essential aspects of code, and recognize unexpected behaviors

Owing to the pervasiveness of abstraction in CT, we feel it is worth looking at the subpractices that define abstraction. For example, the framework of Weintrop *et al.* mentions the practice of creating computational abstractions, which is described as "The ability to conceptualize and then represent an idea or a process in more general terms by foregrounding the important aspects of the idea while backgrounding less important features" [20]. Similarly, in Shute *et al.*'s framework [26], the practice of abstraction contains pattern recognition. In defining this practice, we aim for it to encompass ideas relating to students focusing and planning where to go next.

In the following instance, we present a case where students engage in highlighting and foregrounding. This example takes place near the beginning of a computational activity where students are modeling the motion of a block hanging from a vertical spring. After the students dissect and interpret the different parts of the code, they decide to focus on correctly modeling the graphs of different forms of energy, instead of correctly modeling the different forces in their model.

C: Alright, so we need to add the forces of gravity and the spring, and we need to graph the energies.

A: So which one should we do first?

B: He already gave us the code for the kinetic energy graph, so maybe we should start with the graphs.

C: Okay, so just copy this line? [Student C points to the line of code for the kinetic energy graph.]

This example is a demonstration of highlighting and foregrounding because the students have perceived the different features of the model (i.e., decomposing), and then they decided to focus on coding the different graphs into their model (i.e., highlighting and foregrounding). This leads to the students experiencing difficulty because the graphs will not appear correctly unless the forces have been correctly coded. Regardless, the students could be focusing on this aspect simply because their teacher had already done part of the work for them. Ideally, students would be able to engage in a practice like this on their own. This practice is subtle and difficult to notice, but it is important for computational thinking. As a result, we hope to investigate it more in the future. In many ways, highlighting and foregrounding is a form of planning that the group negotiates among all of its members. For instance, after successfully adding code to add a spring force to their model, the students might go on to focus on graphing all of the energies in their model. Contrastingly, they could choose to focus on different aspects of the code, such as adding friction or changing values in their model to observe the relationships between variables. Most often, highlighting and foregrounding exists as students focus on one component of their model.

C. Translating physics into code—Adapting an analytical model to a computational environment

Literary support for including this CT practice came predominantly from two sources: Weintrop *et al.* and AAPT. Weintrop *et al.*'s framework contains the practice of preparing problems for computational solutions, which is described as reframing problems "so that existing computational tools—be they physical devices or software packages—can be utilized" [20]. Moreover, AAPT's recommendations include a computational physics skill called translating a model into code. This action describes how students should "translate a theoretical or algorithmic model into code that enables computation," which includes constructing readable code, using language documentation, and applying physics knowledge to make decisions [25]. We believe that translating analytical (e.g., written) elements of a problem into code is ubiquitous in computational activities, especially in our context where students are expected to learn with written prompts before modeling physics with the computer.

Translating physics into code can manifest itself in a huge variety of different ways in our context. When students program physics equations in code form, they are directly translating physics concepts into something that can be understood by a computer. An example of this is the Euler-Cromer update equation, which states that a new value is equal to the old value plus a change [35]. This is typically seen when students need to update the position of a moving object. When students are using a simulation to analyze some unknown relationship, they must consider how the computer is going to output the desired results. Much of this practice stems from students consciously deciding how physical relationships are not immediately recognizable to the computer. One cannot simply tell the computer to "exert a force" or "make these objects collide." Instead, one must program the changes by redefining vectors after each iteration throughout a loop.

Below, we provide an example of translating physics into code. Students are trying to modify their code to calculate the elastic potential energy of a vertical spring-mass system. The teacher provided them the correct code to calculate and graph kinetic energy, and now the students are tasked with correctly modeling the graph for elastic potential energy.

A: So I think we need to have, like, the formula for kinetic energy. Oh wait, we already have kinetic energy.

B: Spring energy then?

A: Yeah.

B: Elastic energy equals 0.5 k…

C: Times k…

A: Oh, I forgot that I had to do that. [Student A adds an asterisk between 0.5 and k in their computer code.]

B: Okay now times… um…

A: X squared, right?

C: But we need to use what the code uses instead of x.

B: Yeah, so, times "spring displacement"…

C: Times, again…

A: Oops! Sorry… [Student A adds an asterisk between k and spring displacement in the computer code.]

B: And then put two asterisks and 2.

A: Because it's x squared?

B: Yeah, that's right.

The previous example is a straightforward case of students translating the equation for elastic potential energy into code form. The students identify the equation that they want to translate, and they talk about the equation in code form. Multiple times, student A forgets that she needs to include an asterisk for multiplication between variables. A similar instance occurs when student B instructs his teammate to use two asterisks for carrying out exponential calculations, because Glowscript VPython uses a double asterisk to carry out exponential functions. Furthermore, student C acknowledges that they cannot just use x as the spring's displacement, and instead, they need to use whatever variable name is appropriate for this specific code. By some clever activity design, the MWP provided explicitly defines a variable named spring displacement, and the students use that in their code instead of x. It is important to acknowledge that the computational platform in our context requires students to think carefully about mathematical symbols and variable names when translating physics into code. Ultimately, translating physics into code is specific to introductory physics because it involves bridging the gap between conceptual physics and computational content.

D. Algorithm building—Planning and constructing a series of ordered steps to model a physical phenomenon

Algorithms are commonly referred to when discussing computational thinking. The Shute *et al.* framework gives the most comprehensive coverage with respect to this practice [26]. Although sometimes an entire computer program could be considered an algorithm, we most commonly focus on the example of an algorithm within a position update loop because that is where this practice is most easily observed in our data. This limited view of algorithms might be different from what a computational physicist would commonly think of as an algorithm because it only covers a small range of possible situations when an algorithm would be useful. Owing to our context, we did not observe many of the potential ways that students could engage in algorithm building.

The following transcript is an in-class example of students discussing the algorithmic nature of computer code with their teacher. This students are experiencing the same problem demonstrated in Fig. 3. At this moment, the students are trying to get two box objects to move toward each other simultaneously in the model. Currently, their simulation displays one box moving to the right, and after the first box completes its motion, the second box begins moving to the left (see Fig. 3). The students call their teacher over for help.

T: So you have crate 1 and crate 2…

C: We got them both to move, but it does this…

B: The second one moves after the first one.

T: Oh yeah, so this is a little different. You're familiar with the while loop, right? So think about that. That tells you that while this is true, it's going to do this. So if we think about coding as, like, step-by-step instructions for the computer. You told it to create a box. You told it to create another box. You gave box 1 an initial velocity. Then, you said while this is true, start moving the first box. So when is box 2 going to start moving? Where in your instruction list do you have box 2 moving?

A: Oh, because it's after all this? [Student A points to the while loop, which only updates the position of the first box.] So does it have to be in the same line?

B: Or do we just put it next to it?

T: Okay, so right now crate 2 doesn't get a velocity until after crate 1 moves. [The teacher points to the computer to guide his students' eyes.]

B: So we should put the velocity earlier in the code, like up here, for crate 2?

T: Well, do you want crate 2 to start moving at the same time as crate 1? Yeah, so when you give crate 1 a velocity, give crate 2 a velocity at the same time. [The students look confused.] What I'm saying is take line 19 and put it earlier in your code.

A: So put it first, like before this? [Student A points to above the while loop in their code.]

T: Like at line 14 or 15.

A: Yeah.

C: So then put everything else the same?

T: Sure, we can try that! That will at least get your second box moving at the same time. Basically, it will say step 1, move this box, step 2, move this other box. Instead of move this box, and then once it's done moving, start moving the other box. Think order of operations, like a step-by-step procedure.

After this discussion, the students try to rearrange their code so that both of the crates move in the desired manner. This is a demonstration of algorithm building because the students are interacting with the stepwise nature of computer programming, and they are thinking about the order of commands that need to be executed to correctly model the physics. The teacher sends explicit messages about the sequential nature of programming by dissecting their code and telling them how the computer runs each command. He even describes it as a "step-by-step procedure," and he relates the idea to the mathematical order of operations. After troubleshooting their code for about 5 min, the students cannot figure out how to fix the same problem, so they call the teacher over for help again.

B: It's still moving after the first one.
T: Okay, I'm going to give you a hint on something. Notice how some things are indented? What do you think the indentation means?
C: Oh, yeah, huh… So, indentation means it falls under the while loop.
A: Yeah, it means it does like all those things under the while loop. [Student B deletes lines of code to combine their two separate loops into one loop that will update the positions of both crates at the same time.]
C: Okay, try running it now. [Student B runs the program, and it displays both crates moving toward each other at the same time.]
B: Ah, I see!

This example illustrates many of the different facets of algorithm building. The idea of parallelism (i.e., carrying out steps simultaneously in a computational model) is demonstrated when the group realizes that indented lines after the while statement will be executed together throughout every iteration of the animation loop. Once the students combined their two loops into a single loop that updated the position of both crates, the students realized how to arrange their code to model the two colliding crates correctly. The most straightforward form of this practice occurs when students are considering the control flow of a computer program. Other variants of algorithm building related to efficiency, redundancy, generalizability, and accuracy are expected to occur, and investigation into this area of research could be fruitful.

E. Applying conditional logic—Planning a logical sequence of events and editing conditional statements within a computational problem

Most commonly, the practice of applying conditional logic exists when students are working with while/for logic and if/then/else logic. This practice mostly relates to control flow. Owing to the commonplace nature of looping and conditionals in introductory physics, we value the ability to construct a loop with logical statements. Another argument for its inclusion is that, like if/then/else logic, while/for logic is usually tied to the physics of the model (e.g., position updates within a while loop over time). This is especially true for introductory programming in the Glowscript VPython environment. Computers cannot recognize physical quantities such as "velocity," and it is up to the coder to redefine the code with different conditions.

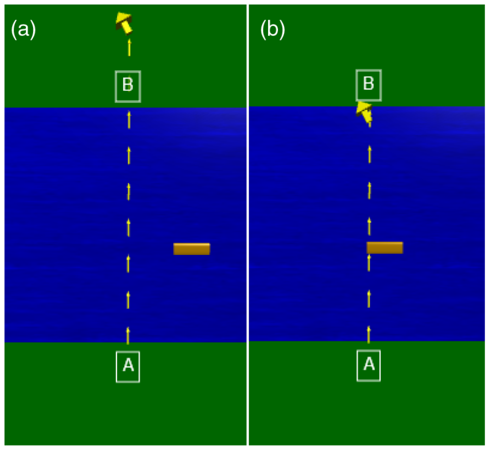

FIG. 4. Computational model of a boat crossing a river. The large yellow arrow represents the boat, the smaller yellow arrows represent a motion map of the boat's velocity at different time instances, and the dark brown rectangle represents a log floating in the river (i.e., only affected by the current). Panel (a) displays the boat incorrectly continuing after it reaches the opposite shoreline, and panel (b) displays the boat correctly stopping when it reaches the opposite shoreline.

In the next segment, students apply conditional logic when modeling the motion of a boat crossing a river with a perpendicular current (see Fig. 4). At this point, the students have successfully added together the vectors for the river's current and the boat's motion such that the boat is traveling in a straight line across the river (i.e., the boat's horizontal velocity component is equal and opposite to the horizontal velocity component of the river). However, the boat continues its motion even after it reaches the opposite shore. The students know that their boat should stop (i.e., the program should stop running, or the position of the boat object should stop updating) side "B." The students focus on the conditions of their while loop in an attempt to achieve this.

D: We should make it not go onto the grass now.
B: Yeah so, uh, if boat position…
D: No, just while "boat.pos.y" is less than something.
B: While… boat.pos.y is less than… [Student B is making changes to the conditional statement of the while loop within the computer code.]
A: Do 120. That's where the B label is at.
B: Oh, really? Okay.

A: Or maybe, like, just before that. We'll see. [Student B runs the program, and it displays the boat moving across the river and still moving beyond the destination shore, as seen in Fig. 4a. Their teacher walks up as they see the outcome of their computational model.]

T: How we doing? What's up?

B: We just basically finished.

C: We're just trying to figure out how to not go in the grass. [Student B changes the value of their while loop to a slightly different value and runs the model again.]

D: I'd say we did it… Well actually, it's still barely going.

B: Okay, how about 85? Let's get some 85 up in here! Run this program. [Student B changes the value in the conditional statement and runs the program again. The model shows the boat stopping as it reaches the shore on the opposite side of the river (Fig. 4b).]

D: Nice! That's good. It stops right when it gets there.

Here, we observe the students applying conditional logic to get their model to stop running at the correct time. They accomplish this goal by changing the conditions of the while loop, which animates the boat's motion depending on its y position. At first, student B thinks about the dilemma with an "if" statement approach, but shortly after this, student D overrules him and decides to change something in the "while" loop. Both of these pathways are valid forms of applying conditional logic. Subsequently, student A proposes using the value of 120 in their conditional statement because that is where the B label is positioned. Using the relative value of one object's position as the animation requirement of another object's motion is a higher level form of applying conditional logic than random guessing and checking. Nevertheless, the students make slight iterative adjustments to the conditional statement in their while loop to get the boat to stop in the right location.

F. Utilizing generalization—Importing previous approaches, algorithms, or code into a model

We chose to name this practice "utilizing generalization," as opposed to just generalization, because we felt that the latter was ambiguous as to whether the practice meant utilizing generalized code or writing code in a general way. Our MWP-based context focuses more on the utilization of existing code. By contrast, one could envision a distinct variant of building computational models in a generalized way, such that the model accommodates a wide range of circumstances. Most MWPs have already made decisions on how the code is structured, and so it is difficult to emphasize coding a model such that it may be useful in future problems. For our context, teachers valued getting students to reuse, remix, and utilize general codes [19]. Thus, our focus is on using general code as a resource for constructing a computational model.

In the next example, students utilize generalization to create a second box object by remixing the working code from the first box within the same program. This occurs through the simple act of copying and pasting one line of code, and then making modifications to the replicated code. The following conversation takes place after the students are tasked with creating a second crate object with different attributes than the first one.

A: Create a second crate with a different size and color, and then place it on the far right side of the floor. [Student A reads the prompt from their worksheet.] Okay, crate equals box… Copy all that. [Student A highlights the entire line of code.] Copy. I'm just going to put it right here. Paste. Okay. So that'll put it…

B: There will be a cube in the same place as the last one.

A: Same exact spot, same exact color, same everything.

C: Well, we need to change the size and color.

A: So let's put the color back to red. Nah, let's do magenta. Magenta actually works, I tried it last time.

B: So magenta, and then change the size. Let's just make it 30. [Student A changes the code accordingly.]

A: Crate equals… Okay, so let's change the position vector to… So, negative 20 puts it on the same axis as the first crate, so let's try positive 50.

B: Let's give this a shot. Boom! [The program displays two crates of different size, color, and x-position.]

A: Wow, that was super easy. Different size, different color, and it's on the far right side. We did it!

This example vividly demonstrates utilizing generalization because the students are copying (i.e., importing) a line of code from elsewhere in their program, and then they are modifying that line to easily create another box object. After reading the worksheet prompt, student A begins by simply copying the code for the first crate and pasting it directly below the original line. The group also acknowledges that the cube will be located at the same position with the same characteristics as the original cube. Consequently, they start the process of remixing the code by changing its color, size, and initial position. By the end of the interaction, student A admits with glee that the entire process was "super easy" because they had experience with utilizing generalization.

G. Adding complexity to a model—Iteratively making computational models more complete, complicated, or realistic by including new physical features

When teaching with MWPs, the models are typically constructed for students, and they only have to add simple features with significant scaffolding. By the time the model is near completion, students should ask themselves, "How can this model be improved?" Students make adjustments that add even more complexity to the model. Examples of these additions include color-coding arrows or objects, adding titles, x and y labels, and legends to a graph, incorporating air resistance or friction, and adding more objects to a model. Sometimes, these improvements are merely cosmetic and they focus on making the model easier to comprehend. Other changes may be extensions of the physics, such as incorporating realistic effects like damping when it was not originally present. The major takeaway is that this process is iterative. Students can run the code, decide if an extension is called for, and then make the necessary edits.

The following example of adding complexity to a model emerges when students are working through an activity that tasks them with modeling the motion of a vertical spring-mass system. Once the students correctly program the graphs for all of energies, they continue to read the next task on their worksheet.

D: In a real situation, the spring would run out of energy. As the spring bounces up and down some of the energy will be lost to the surroundings as thermal energy. Add to your forces a new force called "F damp." This force will need to slow the spring on each successive bounce so that it eventually comes to a stop. The F damp should be proportional to and opposite the velocity. [Student D reads the prompt on their worksheet aloud.]

C: So for F damp, put in negative "cube.velocity," and then we also have to add in the damping coefficient.

A: This is going to be some very thick air. [The other students in the group laugh at student A's joke.]

D: Yeah, now in F net, do all of that plus F damp. Or would it be minus F damp?

A: It's already negative in the line above. [Student A runs the program, and it displays the mass oscillating with the damping force correctly added.]

Even though this case was specifically prompted by the activity's design, the students are still gaining experience with adding complexity to their model. This instructor is asking his students to comprehend that a computational model can always be further developed to more closely match reality. In the end, the students successfully added a damping phenomenon to their model. They do this by translating the equation for the damping force into the code, and then adding the damping force to the net force equation. We see that adding complexity to a model emerges at a very large grain size over a gradual series of interactions between computational thinkers. Over the entire class session, the students successfully added new calculations and graphs of the energies, as well as adding complexity to the physics underlying the model (i.e., adding damping). The idea of making assumptions as a way of simplifying models is paramount to engaging with CT in a physics context.

In general, expanding on a computational model is a complicated process that requires experience and confidence with the computational medium. Because computationally integrated physics courses focus less on providing students experience with programming, they might not have the self-efficacy that allows for effortless communication and advanced ideas around model development. On the other hand, this lessened focus on computer science ideas would be expected to enable students to think more readily about the physics modeled by a simulation. Therefore, adding complexity to a model is a high-level CT practice that can be demonstrated by adding new physical features to a working computational model.

H. Choosing data representation forms—Implementing the best approach, technique, or tool to convey computational results

This practice most closely resembles the data visualization practices in other CT frameworks. Since there are so many methods of producing visualizations with different IDEs (integrated development environments), this practice is bound to occur in almost any computationally integrated science course. When students use Glowscript VPython, the code will almost always produce a visualization. These visualizations are an integral part of computationally integrated physics courses, because they allow the students to visualize motion of objects. This practice places an emphasis on the communicative ability of data visualizations. Students engage in choosing data representation forms when they consider how to make data visualizations more effective at conveying scientific results. Sometimes this could mean creating graphs, printing values, or simply visually inspecting the output of a simulation. The highest form of this CT practice emerges when students are evaluating the visualization for improvement. Ideally, it is up to the students to decide the most illustrative and effective route for presenting their data.

The next example is an illustration of students choosing a data representation form for the quantities in their computational model. After successfully translating the equation for Hooke's law into their model, the simulation displays a vertical spring-mass system with simple harmonic motion. The students continue to add other forms of energy to the graph provided in the MWP. Their teacher designed the program to graph the kinetic energy correctly, and now the students have to model the elastic, gravitational, and total energies.

C: Alright, so right now we have energy versus time. [The program displays a vertical spring-mass system oscillating and the corresponding kinetic energy graph.]

B: It just keeps doing the same thing. There's no friction or anything. Are we going to have to add that later?

D: Yeah, that's what we do at the bottom. [Student D is refers to the last question on their worksheet, which tasks them with adding a damping force to the model.] Right now we have to graph all the energies. Kinetic energy is already done. We need the gravitational potential, the elastic potential, and the total energy.

B: We need to graph those all on the same graph?

D: Yeah.

This example shows the students choosing a data representation form because they realize that the energy versus time data can be effectively visualized with a line graph. Moreover, they decide the data should all be displayed on the same graph, which is a deliberate choice of data representation that will affect the model's clarity. Although the previous example does exhibit students engaging in this practice, it is a more passive form than it would be if the students had actively made some decision for themselves around data representation. Instead, the teacher asks students to engage with this practice by designing the activity so students must graph all the energies. This could be because graphing is a difficult technique to carry out in Glowscript VPython, which might have been these students' first experience with graphing in a computational environment.

I. Generating data—Producing some form of data through the enactment of a computational model

Given the nature of MWP-based computational physics courses, generating data is hard to observe. When the code is ran, data is generated, but the question becomes whether or not the student is consciously considering the visualization output as data generation. Students are not typically running laboratory experiments and using the computer as a mechanism to record raw data. Rather, everything is in the realm of creating data. Students are given an MWP that creates a unique set of data that pertains to one particular physical model. The visual output of a model can serve as a viable form of data, as opposed to more traditional forms of data like graphs or numerical values. Consequently, data generation might look different in contexts that are less visually dependent than Glowscript VPython models.

The following example showcases students generating data when trying to graph the different forms of energy for a vertical spring-mass oscillating system. After the spring force has been correctly coded, the students move on to the next task: adding graphs of kinetic, elastic, gravitational, and total energies. The students begin to discuss what they expect the energies to look like.

C: Kinetic energy is already done.

D: So, when the spring is at its lowest, it will be…

C: The energy will be all elastic…

D: At the end, it will be… Well, go back to the program.

B: For what? Just run it again?

D: Yeah, run it again. [Student B runs the code.] So, that's kinetic energy…

B: So, it's just saying when it's at the bottom, it has zero kinetic energy, which makes sense.

A: Yeah, so what I expect to see for elastic potential is like this… [Student A points her finger in a sinusoidal fashion that is out-of-phase with the kinetic energy.] You know what I mean?

C: Oh yeah. So how do we graph that?

This is a case of generating data because they purposefully run their program to examine the output. They are trying to predict what they expect to see for the graphs of different energies. In many cases, students simply run a computer simulation to gather information and to decide what to do next. Whether students are running code to remind themselves of where they are at, to hash out what they expect to see, or to see if their changes achieved a desired outcome, these are all cases of data generation. A high-level form of this practice exists when students generate data with intentionality. If students specify a reason for running their program, then they are engaging in this practice with purpose, rather than passively generating data by just clicking the run button. When data generation is passive, students are still engaging in the action of generating data, but true computational thinking requires intention.

J. Manipulating data—Preparing for analysis by processing, organizing, and cleaning the dataset

Data manipulation is associated with the handling and managing of data in preparation for further investigation. Datasets can be anything from lists coordinates to vector arrows representing representing physical quantities. When thinking about visual vector arrows as data, the arrows might not be recognizable if they are not scaled properly, despite the physics of the model being correct. Students might use a scaling factor to display the vector arrows on a similar size scale as the rest of the objects in their program. Increasing the size of the arrows does not alter any physical meaning of the model. In this hypothetical case, the data that has been produced (e.g., the vector arrow sizes and directions) needed to be manipulated (i.e., scaled to different sizes) so that the data can be more effectively analyzed. Data manipulation should not alter the physics or the relationships within the data, but rather make these phenomena more apparent.

We were not able to find a presentable example of manipulating data within our video data. The closest case we have is students changing the value of a damping coefficient variable to make the damping more pronounced, but this manipulation actually changes the physics at play in the model. Changing the value of a parameter that significantly alters the physics is not an ideal instance of manipulating data. With that being said, we thought it was necessary to include this practice in our framework, since it is included in other prominent CT frameworks [20]. Although we did not find evidence of this practice in our context, we could still conceive ways that manipulating data occurs. Hopefully, teachers and researchers will have some idea about what to look for when studying manipulating data in the future.

K. Analyzing data—Extracting meaning from a dataset or output of a computational model

Data analysis is the stage in which students will make scientific claims based on what the data is telling them.

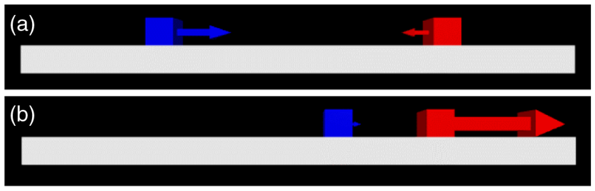

FIG. 5. Computational model of two cubes colliding (a) before and (b) after an elastic collision. The blue cube represents a semi truck and the red cube represents a red Ferrari, with arrows representing each object's velocity.

A computational model is capable of producing data that could provide information to make a claim. Some behaviors that constitute analyzing data include observing relationships, making claims, evaluating the validity of a model, and drawing conclusions. If done thoroughly, the data analysis stage provides evidence of whether or not the scientific hypothesis was true or false. Analyzing data is usually one of the last CT practices that students would engage in before deciding that their computational task is complete.

In the following video example, a group of students is trying model the situation of a head-on collision between a red Ferrari and a blue semi truck from Liam's momentum conservation computational activity. The truck has a mass 4 times that of the Ferrari, and they are moving at different velocities. The computational model displays two cube objects undergoing an elastic collision with conservation of momentum correctly coded (Fig. 5).

A: Alright, let's try this again. [Student A runs the code and it displays the situation depicted in Fig. 5.]

D: Okay, that seems reasonable.

A: So blue barely moved after the collision, and red completely shot off.

D: So the red Ferrari has a higher final velocity because it has less mass?

A: Yeah, I think that's right.

In this clip, we observe the students analyzing the output of their model to draw conclusions about momentum conservation. After entering the real-world parameters provided by their teacher, the students generate data and extract meaning from the data. There are many outcomes that they could focus on, but they realized after the collision, the lower mass object (the red Ferrari) will have a greater final velocity. While this case does illustrate students analyzing data, they could have gone further by using their theoretical physics knowledge (e.g., equations for momentum conservation) to verify the results of their model. Analyzing data tended to occur often, and some of the markers for this practice include students checking to see if their model is behaving as expected, testing a scenario to understand the outcome of a physical situation, or inputting a physics equation to investigate the relationship between variables.

L. Debugging—Remedying unexpected behaviors or error messages when working with computation

Nearly every framework from our review makes mention of troubleshooting associated with CT. Although many frameworks contain a straightforward presentation of debugging, there is still room for interpretation. One example of this is whether or not debugging explicitly includes an error message. Many frameworks include language along the idea of remedying "errors," without specifying if these errors are uniquely code-breaking error messages or, more broadly speaking, any sort of unexpected behavior encountered when developing a computational model. We include any sort of computational issue as something to be remedied through debugging. Owing to the fact that debugging often overlaps with many other practices, it belongs in its own category. For example, a student could engage in either algorithm building or utilizing generalizations while debugging. The student might have referenced a previously completed assignment (one that was generally similar to the current assignment) to fix a bug, or they may have realized that there was a control flow issue in the code's algorithm. This example highlights that debugging is universal, and can occur alongside most CT practices.

When applying debugging to our context specifically, we say that the practice encapsulates everything from the moment an unexpected behavior is identified to when a remedy has been put in place. This not only includes realizing that an error exists but also identifying its location and its source, and deciding on the best solution for the issue. Over time, students will likely have developed a catalog of different computational errors and have ideas on how to resolve them. As for non-error-message unexpected behaviors, the process is a bit more organic. Students have to recognize that an issue is present, which is not always obvious. Then, they have to consider the physics to make a decision on whether or not the model is correct, and they have to figure out where the issue is and why it is happening. If a certain object is moving in an unexpected way, it is likely because the physics that the computer is interpreting is incorrect. One effective approach to diagnosing these types of unexpected behaviors is to question what physical phenomena should be accounting for the unexpected motion, and reviewing how that physics is implemented in the code.

The following example features students debugging a simulation of a hanging spring-mass system. After the students successfully get their spring to oscillate, they are tasked with producing graphs of the different energies of the system. The students just added two lines of code to produce a graphs of elastic potential energy and gravitational potential energy over time. They run the program to see if they correctly programmed the physics, and the program displays Fig. 6(a).

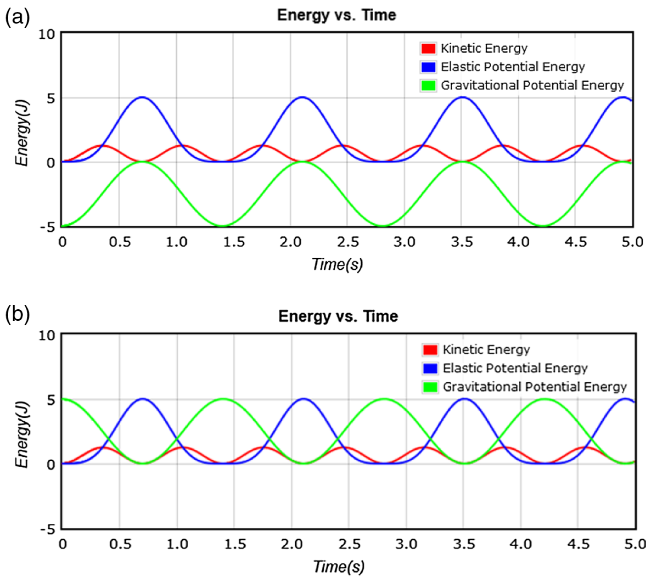

FIG. 6. (a) Incorrect graph and (b) correct graph of energies in a vertical spring-mass system simulation.

A: Oh no! I don't know! [Student A is distressed because of the negative gravitational energy in their graph.]

C: Oh gosh! Energy can't be negative.

A: And it's still in phase…

C: Wait… It needs to be flipped because actually it's just upside down.

A: Yeah, it's a sign error!

B: No, even if you flipped it, the gravitational potential energy wouldn't start at the right spot. It starts high… It should start at zero, right?

C: No, it shouldn't. Gravitational is the most at the beginning because it's all the way wound up.

A: Okay, let's check this… [Student A changes the code, runs the program. The model displays Fig. 6(b).]

The previous transcript provides a case where the program runs without encountering a fatal error, but the students identify an unexpected behavior in the gravitational potential energy. The students express frustration through exclamations of distress ("Oh no!" and "Oh gosh!"). The group is analyzing data to inform them that they need to debug the physics. They know that energy is a scalar quantity, and it should not be negative in this case. The unexpected behavior could be corrected by multiplying by a negative, which is the simplest way to make their model behave in the way they expected. While student A is making changes to the code, student B and C discuss what they expect the value of gravitational energy to be at the beginning of the simulation. We observe that debugging can be a group-oriented process of everyone contributing their ideas.

Troubleshooting attempts do not always end with students being successful. However, as students become more familiar with the coding environment, they could more readily resolve common issues. For example, students would regularly encounter an error where the computer program cannot add a vector and scalar quantity. In many cases, this error can easily be fixed by using a magnitude command [e.g., $KE = 0.5 * \mathrm{mass} * \mathrm{mag}(\mathrm{velocity}) * *2$] to make vector quantities become scalars. After a few experiences with coding in their physics classes, students will immediately use this strategy to resolve the vector or scalar addition error. Debugging is one of the most recurrent CT practices because it applies when either the physics (leading to unexpected behaviors in the model) or the computer code (leading to fatal errors in the computer program) are incorrect. Furthermore, debugging can occur in both systematic (i.e., planned) and nonsystematic (i.e., random or guess-and-check) ways [36]. Although we value systematic debugging as a higher level practice, haphazard debugging can be an effective approach to troubleshooting. Further investigation is needed to explore the variation of frequently occurring CT practices.

M. Working in groups on computational models—Engaging as a member of a team to gain understanding, develop creativity, and complete a computational task

Group work is not exclusive to programming, nor is it easy to classify as a "thinking" practice. However, several of the frameworks that we have drawn upon highlight the social component of computational thinking and provide justifications for its inclusion as a CT practice. Berland and Lee discuss groupwork as distributed computation [22], whereas Brennan and Resnick included "connecting" (i.e., sharing and communicating in computational environments) as a disposition in their framework [22,23]. Groups of people working together is inevitable in the 21st century, and cooperativity is important to CT. In addition, it could be argued that teachers value group work in our context because they are integrating computation with a group-based approach. Initially, the inherent nature of group work in our context was problematic from a research perspective, as we wondered exactly what actions constituted group work. If we simply coded videos for times when students were talking as a group, we would have coded the entire video as working in groups. Instead of this broad coding scheme, we focused on behaviors that required more than one participant. For example, the positioning of the computer could result in both inclusive and exclusionary group dynamics. Encouraging another group member to speak up, or knowing when to reach out to the teacher or peers could also constitute working in groups.

Our video analysis found several instances of disunity (i.e., negative group dynamics) and collaboration (i.e., positive group dynamics). For the following segments, we will provide two examples from Liam's classroom. The first example demonstrates a team of students working in a way that is not conducive to collaborative group work. In

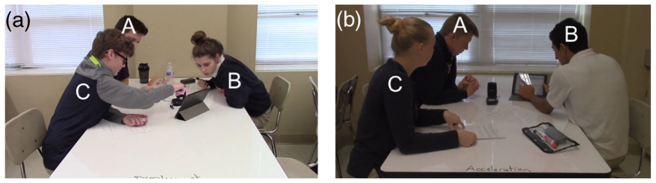

FIG. 7. Images of two contrasting groups working on the colliding crates activity from Liam's classroom. Panel (a) shows students A and C being able to easily see the computer, while student B has to lean in to see the device. Panel (b) shows students A, B, and C all with a line of sight to the computer so that they can easily collaborate.

contrast, the second example will exhibit students working through the same assignment with collaborative group work practices. See Fig. 7 for images of the two contrasting groups. From a cursory observation, group (a) clearly could arrange themselves better to encourage good group work such that student B does not need to lean in to see the computer. On the other hand, group (b) is sitting in a configuration that enables its members to all easily see the computer. The transcripts in the following paragraphs will provide further evidence to support our claims around disunified versus collaborative approaches in computation.

In the first example, we focus on discussing disunity, as it is much easier to observe. This case should serve as a counterexample for what good group work looks like. This video clip is from an early experience with computation in Liam's colliding crates activity. Student C has been told by his instructor that he is not allowed to touch the keyboard because he has much more experience with programming than his group mates.

C: What if we just alternated these commands? So like crate1, crate2, while, while, rate, rate, you know? Just like alternate them…

A: No, I don't know. [Student A sounds defeated.]

B: Yeah, I don't really understand. I feel like I'm going to mess it up. You can just do it. I don't want to mess it up because I don't understand this.

C: I don't really understand what we can do here. It's making me really mad that I can't touch the computer.

B: Wait, so if they have to move at the same time… I don't know, I'm like the worst person to ask about this stuff. I don't get it. [The group members sit silently for about 30 sec.]

A: [Student C] is about to have an aneurysm over here. [Student A uses Student C's real name in the video clip.] He's so mad.

C: I want to touch the keyboard, but I'm not allowed to.

B: Why not?

C: Because he told me I can't. [Student C refers to their teacher with this statement.]

B: Why?

C: I don't know. Because he thinks I would do everything if I did.

At the beginning of the above conversation, student C is trying to guide his group to the correct solution. However, students A and B have such a low self-efficacy with computation, they are unable to make sense of student C's ideas. Student C does not make any attempts to encourage his team members to persevere. Instead, the students just express their discomfort throughout the entire interaction. Student C is frustrated because their teacher, made up a rule that he should not touch the keyboard at all. Liam put this guideline in place so that the other students would have the opportunity to actually do some of the coding themselves. Unfortunately, this approach leads to the group getting frustrated and stagnating. The above example from in-class data demonstrates a snapshot of disunity. The presence of disunity in the majority of groups recorded is illustrative of the need to develop learning goals around computational group work.

The next case shows aspects of positive collaboration when working with computation. Liam's students are trying to fix a bug in their code, and they tended to exhibit positive group interactions (e.g., students sharing the computer amongst evenly, or keeping a positive attitude). This episode of collaboration was captured the same day as the previous example of disunity.

C: Okay, what's with this floor line? Try indenting that line because the box has the size and color under it.

B: Here, you can do it. I don't understand what you mean. [Student C passes the computer to student B.]

C: Like this, so that they're indented. [Student C makes the change to the code and then shows it to her groupmates.] Because when we did the last activity, all the sizes and colors were indented below the first lines.

B: Okay. [Student B runs the program to observe their results, and the program experiences an error immediately upon running.] Unexpected error still…

A: Can I see that real quick? [Student B passes the computer to student A. Student A begins comparing their computer code to the code given by their teacher on the hand-out.] We're missing some stuff. We're missing "$t = t + dt$," and it's indented. It's the last line.

B: $t = t + dt$? No, it's there, but it's not indented.

A: Oh, it is?

B: Yeah, we do have it. See right there? So it should be indented, you think? On a new line?

A: Yup. [Student B takes the computer back.]

B: What are the chances this will work? [Student B runs the program, and it successfully runs without an error.] Hey! It actually worked. Good stuff!

The above situation depicts positive collaboration in a number of ways. First, the students are sharing the computer and the assignment hand-out. They use these resources to collaborate and try new ideas, rather than simply giving up or thinking that they are powerless. Moreover, the students show that they are encouraging and feeling positive about the computation. They reinforce the idea that this is a collaborative space where everyone's ideas are welcome. Overall, student B is encouraging and inquisitive when his group members suggest a new idea. All of the actions discussed here can be indicators of positive collaboration. The previous two episodes provide evidence that shows how having only one computational device or having the coding being dominated by one person can lead to educational inequities.

N. Demonstrating affective dispositions towards computation—Recognizing, expressing, and managing emotions throughout computational thinking

Previously, our teachers indicated they wanted to integrate computation into their curricula to impact students' affect around computation [19]. These teachers are trying to counteract fear and intimidation that computation brings forward in students. These motivators agree with practices outlined in the literature, like Brennan and Resnick's CT dispositions of expressing and persevering. Furthermore, the goal that teachers are proposing can also be related to the idea of computational grit, which has been defined by Duckworth as the "ability to sustain long-term interest in an effort to complete an ongoing task" [37]. The concept of resilience also seems applicable as it is described as having optimism to continue in the face of experienced failures [38]. This connected tree of dispositions highlights the complexity associated with student affect around computation. For our framework, we focus on CT dispositions because there is previous work that has laid the groundwork connecting CT to student affect. Perez refined CT dispositions into three dispositions: (1) tolerance for ambiguity; (2) persistence on difficult problems; and (3) collaboration with others to achieve a common goal [39]. In general, demonstrating affective dispositions can be thought of as recognizing and overcoming the trial-and-error nature of computation.

The following vignette features students demonstrating affective dispositions by persisting with computational problems when working through the colliding crates activity in Liam's classroom (Fig. 8). In this situation, the students' simulation runs, but it is not behaving as they expected (i.e., the students are engaging in debugging).

A: Well, that's not what I expected to happen. [The program displays one box moving across a platform, and then another box appearing after the first box stops.]

B: Not at all. Are they moving at the same rate? Rate 50, rate 50… I don't know. I'm stumped.

A: I don't know what to change. Maybe change that to a negative 35?

B: This one right here? I wonder what that will do. [Student B runs the program, and it remains unchanged.]

C: Maybe we should subtract dt in the $t = t + dt$ line? Like we are counting down to something.

B: So you're saying, make that negative?

C: Yeah, try that.

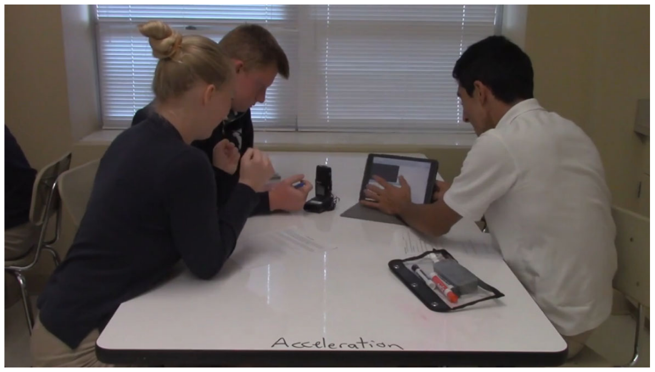

FIG. 8. Image of a group of students experiencing success while working through a computational activity.

B: Oh, okay. Let me just put this back to zero. [Student B undoes the first change.] So $t = t\text{-}dt$?

C: Yeah, try that.

B: That's actually a really good idea! We can try it, maybe… I like where your head is at!

We observe the students making multiple attempts to address an unexpected behavior in their model. Throughout the interaction, student B remains positive and encouraging, and this attitude causes his group mates to feel comfortable suggesting their ideas, even if they may not work. Student B makes multiple utterances that indicate he is maintaining a persistence with solving the problem (e.g., "I wonder what that will do," and "that's actually a really good idea"). Even though the students are stuck on this part of the assignment, they keep a positive attitude. Figure 8 displays the students celebrating after they have successfully debugged their code. The student on the left has her hands raised and everyone is smiling, indicating that they are pleased with their computational model. This could be contrasted with a group that has a negative attitude toward computational material where students are resigned to feeling powerless when working with computation.

Typically, debugging is one of the most frustrating but also one of the more prevalent parts of coding. In turn, evidence of dispositions emerges when students are debugging. This tells us that a deeper exploration of the markers of dispositions needs to be completed. As the teachers indicated, improving attitudes towards computation is an important goal because it teaches students how to persevere through adversity. Although this practice ideally aims for students to have positive affect towards computation, managing emotions of fear, intimidation, and frustration would qualify as this practice.

IV. DISCUSSION

Generally, we observed that activity format, computational platform, and pedagogical approach all influenced how CT emerged in the classroom. For example, when including the instructor as a a part of the group, different

CT practices (or differing quality of the practices) tended to emerge. Keeping track of teacher versus non teacher interactions indicates that students clearly engaged in practices differently when the teacher was present. This finding makes sense, but it also opens up questions about the teacher's role in engaging students with CT practices. Another example would be if computational activities were framed as open-ended modeling activities versus confirmation-style demo activities. In these contrasting cases, we found that different CT practices emerged depending on how the activities were framed.

A consequence of our search for evidence of practices within these learning environments was that we also were able to investigate the emergence, frequency, and relationships between CT practices. While this paper's intention was to focus on providing evidence for the existence of these practices in action, the framework also allows us to measure the frequency of codes emerging. Across different teachers or classrooms, computational activities within one classroom, and groups within one activity we observed differences in CT practices that emerge. Such a top-level analysis provides evidence that when we apply the framework to different classrooms, it can yield different results. At this stage in the research, we still do not know enough to precisely explain the differences, but this tool can be used to explore questions around how curricular design decisions impact the variety and frequency of CT practices.

Some noteworthy trends were identified throughout our video analysis. In general, translating physics into code was the most commonly occurring practice. This is somewhat expected given the nature of the MWP approach. When using such programs, teachers eliminate more complicated computer science ideas while instead prompting students to engage in more physics-related tasks. Algorithm building did not emerge in video data frequently. This result could be either because teachers did not focus on algorithm building as a practice or because algorithm building is a more complex idea embedded within computation. Video data around adding complexity to a model was scarce because this practice tended to occur near the end of activities. For example, if a computer simulation runs with an error message or unexpected behavior, students usually were not focusing on improving the physics of their model. They simply focused on getting the model to work in the first place. Overall, debugging was one of the most frequently emerging CT practices in the videos analyzed. Typically, debugging tended to follow after the data practices because the easiest way for students to identify errors or observe unexpected behaviors was by running the program.

CT practices occur at differing grain sizes. For instance, decomposing may occur at several different grain sizes such as decomposition at the model level or at line-of-code level. Contrastingly, translating physics into code tended to occur at a small grain size which did not frequently overlap with other practices. When translating physics into code, students were usually only thinking and discussing on the level of individual lines of code. Algorithm building occurs at multiple grain sizes. At the largest grain size, the entire computer program can be considered a type of algorithm because it is a series of sequential steps that models a physics scenario. However, there is also a smaller grain size, such as the algorithm of the animation loop itself. Another example is the differing grain sizes of debugging, especially as they pertain to our context, which uses MWPs. When looking specifically at unexpected behaviors, one could argue that the entire process by which an MWP is completed can be thought of as a debugging process; students take an investigative approach to finding bugs and they fix the bugs. Throughout this approach, students might run into small-scale errors as well. Therefore, we acknowledge that these practices occur at different grain sizes, and further investigation is needed to elucidate their variation.

Our investigation found co-occurrence between certain CT practices, while others seemed to be exist exclusively. For instance, decomposing and highlighting or foregrounding were closely related. Students might engage in highlighting and foregrounding at multiple points throughout a computational activity. Often, this practice tended to follow decomposition because after students have interpreted the code, it naturally follows that they will choose a particular piece of the model to focus on after that. Similarly, applying conditional logic tended to overlap with algorithm building because teachers seemed to value the enactment and consequences of loops over simply the sequential nature of coding, and both of these practices work with control flow.

Differences were observed in the CT practices that emerged in Michael's versus Liam's classrooms. While both classrooms were tagged with a similar frequency of codes (approximately 5–10 codes per activity per group over an hour long class session), we observed different CT practices emerging. For example, debugging showed up in Michael's classroom much more than Liam's. In Michael's activities, he would often have students build on the models and change the code significantly, which may result in a greater frequency of debugging. On the other hand, Liam would sometimes use fully working codes to have students engage in more of demonstration-style coding activities. These different curricular choices could lead to an increased presence of debugging. The initial extractions from our analysis demonstrate the applicability of this framework for investigating CT practices in different classroom environments.

Within one classroom, we observed different CT practices emerging over time. This implies that students engage in different CT practices as their experiences evolve over many computational learning activities. For example, in Michael's classroom, the amount of practices appears to increase as students progress through the computational unit. It is possible that as students gain more experience with computation, they are able to engage in more CT practices. Such a result could be used to make claims about how a teacher should design activities as reinforcement exercises rather than learning activities for exploring new conceptual ideas. In a similar way, the framework also yielded different results for differing groups within the same activity. Hence, our CT framework might be used to help teachers design activities and student groups to best fit their specific learning goals.

The work presented in this report aligns well with previous research. First, our framework comprises multiple CT elements nested within several different categories, similar to the structure and approach utilized by Weintrop and coworkers [20]. We provide a taxonomy with clear definitions inspired by past literature. For example, our definition of decomposition utilizes similar language as that proposed by Shute *et al.* [26]. Additionally, some previous research has been done around computational thinking dispositions, and we attempt to capture this aspect of the literature through our inclusion of the “demonstrating affective dispositions towards computation” practice. On the other hand, there are also instances where our framework differs from the literature due to our video analysis results. In the case of debugging, we significantly altered the definition to account for a wider range of scenarios by incorporating the language about “unexpected behaviors” in our definition. Thus, many of our definitions are aligned with other CT researchers, but in some cases, our analysis led to an alteration of previous definitions.

Our work is unique from the current body of literature because our framework has been developed through a hybridized theoretical and empirical approach. Our review of previous research combined with the classroom observations helped us to develop the framework. This is in contrast to many of the previous studies that are cited in past literature. Some of the works emphasized literature reviews [26], expert panels and interviews [23], and classroom artifacts [20]. These pieces of evidence are viable sources of data, but they are limited in scope. On the other hand, our work provides illustrative vignettes with a robust analysis to fully explore students’ actions and speech in computationally integrated physics classes. In doing this work, we provide actionable definitions and examples of CT that are grounded in the behaviors of students in real physics classrooms. This blended theoretical and empirical approach is unique to the field of physics education research. Our data collection techniques and video analysis provide a solid base to further explore CT in computationally integrated physics courses.

Another distinction between our work and previous research is our emphasis on context and how it impacts computational thinking. Throughout this entire report we have stressed the importance of contextual factors and their effect on students’ experiences with computation. Other literature, like that of Weintrop *et al.*, is designed to be universal across mathematics and science classrooms [20]. As a result, they provide a good overview of the general skills that constitute computational thinking. However, it is difficult to apply such a broad framework in a practical way. On the other hand, our work focuses specifically on students in physics classes coding with VPython. Our discussion provides a rich analysis of frequency, grain size, relationships, and other factors influencing the emergence of CT practices. We believe that for a framework to be actionable, it should have clear examples and definitions aimed at a particular setting. In our future work, we intend to more deeply analyze the relationships between curricular design features and computational thinking. Hence, the development of this framework is heavily influenced by our context of introductory physics.

V. CONCLUSIONS

In conclusion, this research began with the question, “How do computational thinking practices manifest in introductory physics classrooms?” Our results in Sec. III exemplify the variation of CT practices that emerged when students worked through these computational activities. Furthermore, our Sec. IV provides a deep explanation of how these practices arose. Numerous contextual factors including the activity format, computational platform, and group composition influenced the manifestation of CT in our data. Most interestingly, different CT practices emerged depending on the role of the instructor. The more involved that the instructor was with their students, the easier it became to identify high-quality instances of computational thinking. Lastly, the frequency and co-occurrence of practices was also influenced by different pedagogical choices. For example, demonstration-style simulation activities caused students to encounter less instances of debugging compared to activities where students were tasked with constructing computational models. Therefore, the work presented in this report serves as a robust investigation of the manifestation of CT in introductory physics classrooms.

This study presents some limitations that should be carefully considered when studying CT practices in other contexts. Our literature review did not include the entirety of papers written on the topic of CT. Instead, we considered sources that focused on frameworks with clearly categorized practices that were relevant to the context of introductory physics. Some of these sources include ideas that are akin to CT without actually being called CT, like where the AAPT source describes these ideas as computational physics skills or technical computing skills [25]. Another limitation is that we did not include some of the more advanced ideas that were discussed in other frameworks such as systems thinking. We chose not to keep some of these more advanced ideas because they were either not

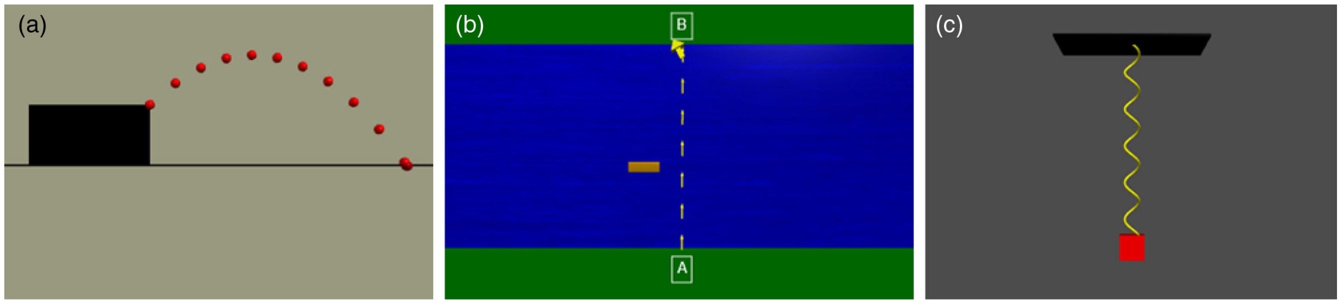

FIG. 9. Outputs displayed in Michael's solutions for computational models of (a) projectile, (b) river crossing, and (c) spring energy problems.

relevant or too advanced for the context that we are interested in exploring.

Introductory physics is a discipline well suited to explore computational thinking practices because physics is becoming increasingly computational in nature and students are being encouraged to learn programming ideas in core science disciplines. Science standards and recommendations from professional physics organizations provide motivation for teachers to integrate computation into their courses. With the lack of clear guidelines around computational thinking, a CT framework like the one presented in this report, can serve as a tool for further examining student practices in computationally integrated physics courses. This framework will help teachers by giving examples of indicators of CT practices. This framework will also be useful for researchers who would like to investigate how various factors (e.g., activity design, student group composition, messaging from the teacher) can lead to the emergence of different CT practices. Our future work hopes to address the variation observed within each CT practice and provide assessment strategies for CT in introductory physics contexts. While there is more to be understood about CT practices, the framework presented in this report is a jumping off point for more closely examining CT in introductory physics.

ACKNOWLEDGMENTS

We thank the National Science Foundation (DRL-1741575) for funding. We also acknowledge Joe Krajcik, Bob Geier, and Sue Carpenter for allowing us to use the CREATE for STEM facilities.

APPENDIX A: DETAILS ABOUT BOTH CLASSROOMS STUDIED

1. Classroom 1: Michael's AP Physics classroom

The first classroom, taught by Michael, was an AP physics C (i.e., mechanics) course for seniors. In this classroom, computational activities took place every Friday for the whole school year. Students were assigned to groups of three to five students, and the group all shared one desktop computer to build computational models on the Glowscript platform. Typically, Michael's MWPs started with a model that would run without error but would not model some aspect of the physics correctly. These activities were often used as supplements to other lab or lecture activities that students worked with earlier in the week. Michael's activity prompts gave students guidance on how to complete the activities, without directly instructing them how to modify the code (e.g., "you'll have to write lines of code to designate the forces on the block; Fgrav and Fspring").

The three activities examined from Michael's classroom were a projectile problem, a river crossing problem, and a spring energy problem which can be found in Appendix 2 (see Fig. 9). In the projectile problem, depicted in Fig. 9a, the students are tasked with changing the initial conditions of the model (cliff height, initial ball height, initial velocity, and angle), modifying a while loop to make the program stop when the ball hits the ground, and displaying the final results (horizontal distance, max height, and final velocity) with a print statement. This computational activity was used to model an experimental projectile motion lab that the students did earlier in the week.

For the river crossing problem, shown in Fig. 9(b), the students first calculate (on a whiteboard) the angle that a boat must move in to reach the other side of a river straight across from where it started. Subsequently, the students work with a computational model to simulate the scenario that they calculated on the whiteboard. To complete the computational portion, the students have to change the boat's angle and modify the boat's velocity by adding the velocity of the river's current to the velocity of the boat.

Michael's spring energy problem features a red block hanging on a yellow spring by a black support. The students have 3 primary tasks to complete this activity. First, the students need to get the block to bounce up and down by adding equations for the forces of gravity and the spring. Second, the group is tasked with adding graphs for gravitational, elastic, and total energy values; they are given a working graph of kinetic energy by Michael in the MWP. Lastly, if the students make it this far, they are prompted to add a damping force to their computational model. Overall, this activity was one of the more involved activities of

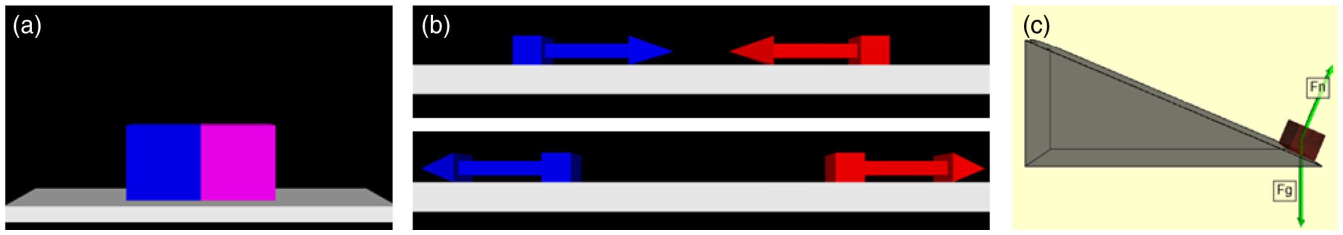

FIG. 10. Outputs displayed in Liam's solutions for computational models of (a) colliding crates, (b) momentum conservation (top panel for before the collision and bottom panel for after the collision), and (c) block on a ramp activities.

Michael's classroom, which makes sense as it was positioned near the end of the students' second semester of working with computation.

2. Classroom 2: Liam's Physical Science 2 classroom

The second classroom examined in this work was Liam's physical science 2 advanced classroom for sophomore students. In this classroom, computational activities were scattered into the curriculum about once every month, giving the students about 4 total experiences with computational modeling per semester. Students were assigned to groups of three to five students, and they all shared one school-provided laptop per group. Students also worked at desks that had whiteboard tops to facilitate easy writing and sharing of ideas. Liam's computational activities were treated as introductory experiences to programming, rather than a reinforcement of physics concepts or labs experienced earlier in the course. This meant that students were usually learning how to interpret code, modify simple aspects of a computational model, and test or predict the outcome of scenarios with their simulation. Typically, Liam's students were given fully working codes, and the worksheets provided to students asked questions that needed to be answered directly by interpreting the code. First, students were given a day to simply change code in Glowscript to spell their name by positioning objects and changing their attributes (e.g., size, position, angle). Our analysis focuses on three activities, following the aforementioned introductory activity from Liam's classroom: 1D motion colliding crates, 1D momentum conservation, and a block on a ramp. Figure 10 displays the output of computational models for activities from Liam's classroom.

After their first experience with computation, students in Liam's classroom work with a computational model of two crates moving toward each other. The final output of the colliding crates activity is displayed in Fig. 10(a). In the MWP, Liam provides his students with the correct code to model 1D motion of one crate moving across a floor object. Students are tasked with changing some parameters of the model (e.g., "Give the crate a different constant velocity moving to the right."), interpreting the code (e.g., "Which lines of code make the crates move?"), adding to the model (e.g., "Create a second crate with a different size, color, and position), and getting the simulation to stop when the two crates meet in the center (e.g., "Figure out how to stop the program once the crates collide."). This activity provides students with a more guided, in-depth activity to work with computational modeling and thinking practices.

The second computational activity analyzed from Liam's classroom was the momentum conservation activity [see Fig. 10(b)]. To give students experience with building on previous models, the computational model of momentum conservation used a code similar to the previous colliding crates activity. Liam provided his students with a worksheet that comprised several phases: analyzing the code, playing around with the model, making predictions, modifying the code, and testing predictions. Students were tasked with explaining what different parts of the code meant to them, changing the code to observe the effect of their changes, and testing or predicting the outcome of different scenarios when plugging in specific real-world values. In this activity, Liam emphasizes the utility in being able to rapidly test different situations with a computational model. By the end of the momentum conservation activity, students used their code to model elastic collisions between two boxes, a car and a truck, and a dodgeball and a teacher.

The block on a ramp problem in Liam's class was students' final experience with computational modeling in the physical science course. Figure 10(c) illustrates the final output of the block on a ramp computational model. The MWP given to students begins with a block sliding down a ramp with the normal force and gravity force vectors depicted by green arrows. First, students are tasked with extracting important details from the code (e.g., "What is the angle of the ramp?"). Next, the students had to make changes to the code and observe the effect (e.g., "What happens if you increase the angle of the ramp?"). Lastly, if the students got far enough, students were instructed to add friction to their simulation. In the end, this activity was a more involved and complicated activity than the previous two assignments.

APPENDIX B: FULLY WORKING CODES FOR COMPUTATIONAL ACTIVITIES STUDIED

1. Michael's projectile motion computational model

```
GlowScript 3.0 VPython
scene.range = 100
scene.center = vector(0,0,0)
scene.width = 960
scene.height = 400
scene.background = vector(0.6,0.6,0.5)

cliffheight = 25
cliff = box(pos=vector(-25,cliffheight/2,0),
size = vector(50,cliffheight,0), color = vector(0,0,0))

ball = sphere(pos = vector(0,cliffheight,0), radius = 2,
        color = vector(1,0,0), texture = textures.rough,
        make_trail = True, trail_type = "points",
        trail_radius = 2, interval = 10, retain = 50)

v0 = 29
angle = 43.6
anglerad = angle*3.14/180

v1 = vector(-100,0,0)
v2 = vector(200,0,0)
ground = curve(pos = [v1,v2],color = vector(0,0,0))
vx = v0*cos(anglerad)
v0y = v0*sin(anglerad)
ballvelocity = vector(vx,v0y,0)
g = vector(0,-10,0)

t = 0
tf = 100
dt = 0.05

timetotop = v0y/mag(g)
maxheight = 0.5*mag(g)*(timetotop)**2+
        v0y*timetotop

while t > 0:
rate(100)

ball.pos = ball.pos+ballvelocity*dt
ballvelocity = ballvelocity+g*dt

t = t + dt

print("vx =",vx)
print("v0y =",v0y)
print("Time to Max Height =",timetotop)
print("Max Height =",maxheight)
print("Time to Ground =",t)
print("Horizontal Range =",round(ball.pos.x))
```

2. Michael's river crossing computational model

```
GlowScript 3.0 VPython
scene.center = vector(0,255,0)
scene.range = 480
scene.width = 640
scene.height = 500
scene.background = vector(0,0.4,0)

currentspeed = 8
boatspeed = 17
boatangledeg = 28.1
boatanglerad = boatangledeg*3.14/180

boat = arrow(pos = vector(0,0,0), color = vector(1,1,0),
        axis = vector(-40*sin(boatanglerad),
        40*cos(boatanglerad),0), shaftwidth = 5,
        headwidth = 15)
water = box(pos = vector(0,255,0), color = vector(0,0,1),
        size = vector(1280,510,2))
log = cylinder(pos = vector(20,255,10),
color = vector(0.7,0.5,0), size = vector(60,20,4))

A = label(text = 'A', pos = vector(0,-30,0), align =
'center', color = vector(1,1,1))
B = label(text = 'B', pos = vector(0,540,0), align =
'center', color = vector(1,1,1))

t = 0
tf = 100
dt = 0.01

currentvelocity = vector(currentspeed,0,0)
logvelocity = vector(currentspeed,0,0)
boatvelocity = vector(-boatspeed*sin(boatanglerad),
  boatspeed*cos(boatanglerad),0)
resultantvelocity = boatvelocity + currentvelocity

cts = label(pos = vector(300,600,0), text = 'Click to
  Start', space = 30, height = 16, border = 5, font = 'sans',
  box = False)
clicktostart = scene.waitfor('click')

while boat.pos.y <= 510:
rate(500)

boat.pos = boat.pos + resultantvelocity*dt
log.pos = log.pos + logvelocity*dt

t = t + dt

print("Time =", t)
print("Resultant velocity =", mag(resultantvelocity))
```

3. Michael's spring energy computational model

```
GlowScript 2.7 VPython
scene.center = vector(0,-0.20,0)
scene.range = 1.2
scene.width = 640
scene.height = 400
scene.background = vector(0.3,0.3,0.3)

support = box(pos = vector(0,0.25,0), size = vector
(0.80,0.01,0.4), color = vector(0,0,0))

initialcubepos = 0
springdisplacement = 0

cubeside = 0.10
cube = box(pos = vector(0,initialcubepos,0), size =
vector(cubeside,cubeside,cubeside), color = vector(1,0,0))
cubemass = 0.5

spring = helix(pos = vector(0,support.pos.y,0), axis =
vector(0,cube.pos.y-support.pos.y,0), radius = 0.03, coils
= 5.5, thickness = 0.01, color = vector(1,1,0))
```

```
cubevelocity = vector(0,0,0)
g = vector(0,10,0)
k = 10
c = 0.5

zeroheight = 2*cubemass*mag(g)/k

t = 0
tf = 5
dt = 0.0001

xtgraph = graph(width = 640, height = 250,
title = ’Energy vs. Time’,xtitle = ’Time (s)’,
ytitle = ’Energy (J)’,
foreground = color.black, background = color.white,
xmin = 0, xmax = 5, ymin = -5, ymax = 10)

kineticcurve = gcurve(color = vector(1,0,0),
label = “Kinetic Energy”)
elastpotcurve = gcurve(color = vector(0,0,1),
label = “Elastic Potential Energy”)
gravpotcurve = gcurve(color = vector(0,1,0),
label = “Gravitational Potential Energy”)

while t < tf:

rate(5000)
Fgrav = cubemass*g
Fspring = -k*cube.pos
Fdamp = -c*cubevelocity
Fnet = -Fgrav+Fspring+Fdamp

cubeacceleration = Fnet/cubemass

cube.pos = cube.pos+cubevelocity*dt
cubevelocity = cubevelocity+cubeacceleration*dt
spring.axis = spring.axis+cubevelocity*dt

springdisplacement = abs(cube.pos.y)

kineticenergy = 0.5*cubemass*mag(cubevelocity)**2
elastpotenergy = 0.5*k*mag(cube.pos)**2
gravpotenergy = cubemass*mag(g)*(zeroheight-spring
displacement)
totenergy = kineticenergy+elastpotenergy+gravpotenergy

kineticcurve.plot(t,kineticenergy)
elastpotcurve.plot(t,elastpotenergy)
gravpotcurve.plot(t,gravpotenergy)

t = t + dt
```

4. Liam’s colliding crates computational model

```
GlowScript 3.0 VPython
scene.width = 1600
scene.height = 600

floor = box(pos = vector(0,-30,0), size = vector
(100,4,12), color = color.white)
crate = box(pos = vector(-35,-18,0), size = vector
(20,20,5), color = color.blue)
crate2 = box(pos = vector(35,-13,0), size = vector
(20,30,5), color = color.magenta)

t = 0
tf = 0.940
dt = 0.01

cratev = vector(10,0,0)
crate2v = vector(-10,0,0)

while crate.pos.x < 35:
rate(300)
if crate.pos.x+crate.size.x/2 >=
crate2.pos.x-crate2.size.x/2:
cratev.x = 0
crate2v.x = 0
crate.pos = crate.pos+cratev*dt
crate2.pos = crate2.pos+crate2v*dt
t = t + dt
```

5. Liam’s momentum conservation computational model

```
GlowScript 2.7 VPython
BlueBox = box(pos = vector(-6,1,0), length = 1,
width = 1, height = 1, color = color.blue)
RedBox = box(pos = vector(6,1,0), length = 1, width = 1,
height = 1, color = color.red)
Ground = box(pos = vector(0,0,0), length = 20,
width = 1, height = 1, color = color.white)

BlueBox.velocity = vector(10,0,0)
RedBox.velocity = vector(-10,0,0)
BlueBox.mass = 10
RedBox.mass = 10
BlueBox.momentum = BlueBox.mass * BlueBox.velocity
RedBox.momentum = RedBox.mass * RedBox.velocity
ScalingFactor = 0.5

BlueBoxVArrow = arrow(pos = BlueBox.pos,
axis = BlueBox.velocity*ScalingFactor, color = BlueBox
.color)
RedBoxVArrow = arrow(pos = RedBox.pos,
axis = RedBox.velocity*ScalingFactor, color = RedBox
.color)

dt = 0.01, t = 0, tf = 1.0

while t < tf:
rate(50)

NextPos_BlueBox = BlueBox.pos+BlueBox.velocity*dt
NextPos_RedBox = RedBox.pos+RedBox.velocity*dt

if mag(NextPos_RedBox-NextPos_BlueBox) <
    (BlueBox.length/2+RedBox.length/2):

VelocityAfterCollision_BlueBox = ((BlueBox.mass-
    RedBox.mass)*BlueBox.velocity+
    2*RedBox.mass*RedBox.velocity) / (RedBox.mass
    +BlueBox.mass)
```

```
VelocityAfterCollision_RedBox = (2*BlueBox.mass
    *BlueBox.velocity+(RedBox.mass- BlueBox.mass)
    *RedBox.velocity) / (BlueBox.mass+RedBox.mass)

BlueBox.velocity = VelocityAfterCollision_BlueBox
RedBox.velocity = VelocityAfterCollision_RedBox
BlueBox.pos = BlueBox.pos+BlueBox.velocity*dt
RedBox.pos = RedBox.pos+RedBox.velocity*dt
BlueBoxVArrow.pos = BlueBox.pos
BlueBoxVArrow.axis = BlueBox.velocity*ScalingFactor
RedBoxVArrow.pos = RedBox.pos
RedBoxVArrow.axis = RedBox.velocity*ScalingFactor

t = t + dt
```

6. Liam's block on ramp computational model

```
GlowScript 3.0 VPython

scene.range = 15
scene.background = vector(1, 1, 0.8)

ramp_angle=23 * pi/180
ramp_depth = 6
ramp_width = 30
ramp_position = vec(-15, -10, 0)

ramp_color = color.gray(0.3)
edge_thickness = 0.05
edge_color = color.black

a = vertex(pos = vec(0, 0, 0)+ramp_position,
    color = ramp_color, opacity = .5)
b = vertex(pos = vec(0, 0, ramp_depth)+ramp_position,
    color = ramp_color, opacity = .5)
c = vertex(pos = vec(ramp_width, 0, 0)+ramp_position,
    color = ramp_color, opacity = .5)
d = vertex(pos = vec(ramp_width, 0, ramp_depth)
    +ramp_position, color = ramp_color, opacity = .5)
e = vertex(pos = vec(0, ramp_width * tan(ramp_angle),
    0)+ramp_position, color = ramp_color, opacity = .5)
f = vertex(pos = vec(0, ramp_width * tan(ramp_angle),
    ramp_depth)+ramp_position, color = ramp_color,
    opacity = .5)

ramp_base = quad(v0 = a, v1 = b, v2 = d, v3 = c)
ramp_surface = quad(v0 = e, v1 = f, v2 = d, v3 = c)
ramp_back = quad(v0 = e, v1 = f, v2 = b, v3 = a)
ramp_side_near = triangle(vs = [a, c, e])

ramp_side_far = triangle(vs = [b, d, f])
def edge(v1, v2, r):
cylinder(pos = v1.pos, axis = v2.pos-v1.pos, radius = r,
    color = edge_color)

ab = edge(a, b, edge_thickness)
ac = edge(a, c, edge_thickness)
bd = edge(b, d, edge_thickness)
cd = edge(c, d, edge_thickness)
ae = edge(a, e, edge_thickness)
bf = edge(b, f, edge_thickness)
ef = edge(e, f, edge_thickness)
df = edge(d, f, edge_thickness)
ec = edge(e, c, edge_thickness)

block_width = 3
block_height = 3
block_depth = 3

block = box (pos = vector(0.5*(block_width**2+
    block_height**2)**0.5 * cos(atan(block_height
    /block_width) - ramp_angle),
    ramp_width * tan(ramp_angle)+
    0.5*(block_width**2 +block_height**2)**0.5 *
    sin(atan(block_height/block_width) - ramp_angle),
    0.5*ramp_depth)+ramp_position,
axis = vector(ramp_width, -ramp_width
    * tan(ramp_angle), 0),
size = vector(block_width,block_height,block_depth),
color = vector(0.65, 0.15, 0.15),
texture = textures.wood_old,
opacity = 0.7)

mblock = 50
vblock = vector(0, 0, 0)
g = vector(0,-9.8,0)
t = 0
tf = 1
dt = .001

Fgrav = mblock * g
Fnorm = vector(mag(Fgrav) *
cos(ramp_angle)*sin(ramp_angle), mag(Fgrav)
*cos(ramp_angle)*cos(ramp_angle), 0)
Fnet = Fgrav+Fnorm

FgravArrow = arrow(pos = block.pos,
    axis = Fgrav/mblock, shaftwidth = 0.3, color =
    color.green)
FnormArrow = arrow(pos = block.pos,
    axis = Fnorm/mblock, shaftwidth=0.3, color =
    color.green)
FgravLabel = label(pos = FgravArrow.pos, text = 'Fg',
    xoffset = -20, yoffset = -50, space = 30, height = 16,
    border = 4, font = 'sans', line = False, color =
    color.black)
FnormLabel = label(pos = FnormArrow.pos, text = 'Fn',
    xoffset = 20, yoffset = 50, space = 30, height = 16,
    border = 4, font = 'sans', line = False, color =
    color.black)

while block.pos.y > (ramp_position.y+
0.5*(block_width**2+block_height**2)**0.5 *
cos(atan(block_width/block_height) - ramp_angle)):
rate(999)

block.pos = block.pos+vblock*dt+
0.5*(Fnet/mblock)*dt**2

FgravArrow.pos = FgravArrow.pos+vblock*dt+
    0.5*(Fnet/mblock)*dt**2
```

```
FnormArrow.pos = FnormArrow.pos+vblock*dt+
    0.5*(Fnet/mblock)*dt**2
FgravLabel.pos = FgravLabel.pos+vblock*dt+
    0.5*(Fnet/mblock)*dt**2
FnormLabel.pos = FnormLabel.pos+vblock*dt+
    0.5*(Fnet/mblock)*dt**2
vblock = vblock+(Fnet/mblock)*dt
t = t + dt
```

[1] D. Mohaghegh and M. McCauley, Computational thinking: The skill set of the 21st century, Int. J. Comput. Sci. Inf. Technol. **7**, 1524 (2016), https://hdl.handle.net/10652/3422.

[2] J. M. Wing, Computational thinking, Commun. ACM **49**, 33 (2006).

[3] J. M. Wing, Computational thinking's influence on research and education for all, Italian J. Educ. Technol. **25**, 7 (2017).

[4] S. Bocconi, A. Chioccariello, G. Dettori, A. Ferrari, K. Engelhardt, P. Kampylis, and Y. Punie, Developing computational thinking in compulsory education—implications for policy and practice, Joint Research Centre (JRC), 1 (2016).

[5] C. Brackmann, D. Barone, A. Casali, R. Boucinha, and S. Muñoz-Hernandez, Computational thinking: Panorama of the Americas, 2016 International Symposium on Computers in Education, *SIIE 2016: Learning Analytics Technologies*, 1 (2016), 10.1109/SIIE.2016.7751839.

[6] H. J. So, M. S. Y. Jong, and C. C. Liu, Computational thinking education in the Asian pacific region, Asia-Pac. Educ. Researcher **29**, 1 (2020).

[7] D. Barr, J. Harrison, and L. Conery, Computational thinking: A digital age skill for everyone, Learning Leading with Technol. **38**, 20 (2011), https://eric.ed.gov/?id=EJ918910.

[8] C. Sneider, C. Stephenson, B. Schafer, and L. Flick, Exploring the science framework and the NGSS: Computational thinking elementary school classrooms, Science and Children **52**, 10 (2014), https://www.proquest.com/openview/12c0a6c4a6f7cf012e173c3c8bc322bb/1?pq-origsite=gscholar&cbl=41736.

[9] N. R. Council, *A Framework for K–12 Science Education* (National Research Council, Washington, DC, 2012).

[10] Partnership for Integrating Computation into Undergraduate Physics, https://www.compadre.org/PICUP/ (2020).

[11] S. Brophy, M. Caballero, K. Fisler, M. Hicks, R. Hilborn, C. M. Romanowicz, K. Roos, and R. Vieyra, *Advancing Interdisciplinary Integration of Computational Thinking in Science* (American Association of Physics Teachers Conference Report, College Park, MD, 2020).

[12] P. Sengupta, J. S. Kinnebrew, S. Basu, G. Biswas, and D. Clark, Integrating computational thinking with K–12 science education using agent-based computation: A theoretical framework, Educ. Inf. Technol. **18**, 351 (2013).

[13] C. M. Orban and R. M. Teeling-Smith, Computational thinking in introductory physics, Phys. Teach. **58**, 247 (2020).

[14] O. Sand, T. O. Odden, C. Lindstrøm, and M. Caballero, How computation can facilitate sensemaking about physics: A case study, presented at PER Conf. 2018, Washington, DC, 10.1119/perc.2018.pr.Sand.

[15] S. Basu, K. W. McElhaney, S. Grover, C. J. Harris, and G. Biswas, A principled approach to designing assessments that integrate science and computational thinking, Proc. Int. Conf. Learn. Sci., ICLS **1**, 384 (2018), https://repository.isls.org//handle/1/819.

[16] M. D. Caballero, M. A. Kohlmyer, and M. F. Schatz, Fostering computational thinking in introductory mechanics, AIP Conf. Proc. **1413**, 15 (2012).

[17] D. Weintrop, E. Beheshti, M. S. Horn, K. Orton, L. Trouille, K. Jona, and U. Wilensky, Interactive assessment tools for computational thinking in high school STEM classrooms, Lecture Notes Institute Comput. Sci. Social-Informatics Telecom. Engin. **136**, 22 (2014).

[18] H. Swanson, G. Anton, C. Bain, M. Horn, and U. Wilensky, *Computational Thinking Education* (Springer, Singapore, 2019).

[19] D. P. Weller, M. D. Caballero, and P. W. Irving, Teachers' intended learning outcomes around computation in high school physics, presented at PER Conf. 2019, Provo, UT, 10.1119/perc.2019.pr.Weller.

[20] D. Weintrop, E. Beheshti, M. Horn, K. Orton, K. Jona, L. Trouille, and U. Wilensky, Defining computational thinking for mathematics and science classrooms, J. Sci. Educ. Technol. **25**, 127 (2016).

[21] V. Barr and C. Stephenson, Bringing computational thinking to K–12: What is involved and what is the role of the computer science community?, ACM Inroads **2**, 48 (2011).

[22] M. Berland and V. R. Lee, Collaborative strategic board games as a site for distributed computational thinking, Int. J. Game-Based Learn. **1**, 65 (2011).

[23] K. Brennan and M. Resnick, New frameworks for studying and assessing the development of computational thinking, in *Proceedings of the 2012 Annual Meeting of the American Educational Research Association* (2012), Vol. 1, p. 135, https://www.media.mit.edu/publications/new-frameworks-for-studying-and-assessing-the-development-of-computational-thinking/.

[24] N. M. Hutchins, G. Biswas, M. Maróti, Á. Lédeczi, S. Grover, R. Wolf, K. P. Blair, D. Chin, L. Conlin, S. Basu, and K. McElhaney, C2STEM: A system for synergistic learning of physics and computational thinking, J. Sci. Educ. Technol. **29**, 83 (2020).

[25] AAPT Undergraduate Curriculum Task Force, *AAPT Recommendations for Computational Physics in the Undergraduate Physics Curriculum* (American Association of Physics Teachers, College Park, MD, 2016).

[26] V. J. Shute, C. Sun, and J. Asbell-Clarke, Demystifying computational thinking, Educ. Res. Rev. **22**, 142 (2017).

[27] J. Lyon and A. Magana, The use of engineering model-building activities to elicit computational thinking: A design-based research study, J. Engin. Educ. **110**, 184 (2021).

[28] T. Palts and M. Pedaste, A model for developing computational thinking skills, Informatics Educ. **19**, 113 (2020).

[29] VPython, https://vpython.org/.

[30] S. Weatherford and R. Chabay, Student predictions of functional but incomplete example programs in introductory calculus-based physics, AIP Conf. Proc. **1513**, 42 (2012).

[31] R. Boyatzis, *Transforming Qualitative Information: Thematic Analysis and Code Development* (Sage Publications Inc., 1998).

[32] V. Braun and V. Clarke, Using thematic analysis in psychology, Qualitative Res. Psychol. **3**, 77 (2006).

[33] R. E. Scherr, Gesture analysis for physics education researchers, Phys. Rev. ST Phys. Educ. Res. **4**, 010101 (2008).

[34] V. K. Otero, D. B. Harlow, and D. B. Harlowe, Getting Started in Qualitative Physics Education Research, Rev. Perinat. Med. **2**, 1 (2009), https://www.compadre.org/per/items/detail.cfm?ID=9122.

[35] A. Chromer, Stable solutions using the euler approximation, Am. J. Phys. **49**, 455 (1981).

[36] M. J. Obsniuk, P. W. Irving, and M. D. Caballero, A Case Study: Novel Group Interactions through Introductory Computational Physics, presented at PER Conf., College Park, MD 10.1119/perc.2015.pr.055.

[37] A. L. Duckworth and P. D. Quinn, Development and validation of the short Grit Scale (Grit-S), J. Personality Assess. **91**, 166 (2009).

[38] K. Miller, 5+ Ways to Develop a Growth Mindset Using Grit and Resilience (2021), https://positivepsychology.com/5-ways-develop-grit-resilience/.

[39] A. Pérez, A framework for computational thinking dispositions in mathematics education, J. Res. Math. Educ. **49**, 424 (2018).