JOURNAL OF AEROSPACE INFORMATION SYSTEMS
Vol. 11, No. 10, October 2014

Check for updates

# Education in the Crosscutting Sciences of Aerospace and Computing

Ella M. Atkins*
*University of Michigan, Ann Arbor, Ann Arbor, Michigan 48109*

DOI: 10.2514/1.I010193

**The aerospace engineering pillars of structures, gas dynamics, and dynamics and control have rested long term on a firm foundation of calculus-based physics. Today's aerospace engineer relies more on the computational device than the pencil and paper to innovate in theory and in practice. Educators and practitioners struggle with questions in how to improve computing knowledge and skills in the aerospace curriculum. Whereas most agree it would be unwise to compromise most content currently in place, opinions differ on which computing concepts students need to know. The aim of this paper is to highlight the importance of computing for aerospace and encourage critical thought about options for its inclusion into the required undergraduate curriculum. To do this, computer science is defined and common misconceptions are discussed. A review of computing content in aerospace curricula is presented. A pedagogical strategy that capitalizes on analogs between discrete and continuous-valued physical processes is proposed, including examples of how computational thinking can be meaningfully infused without loss of content in traditional aerospace courses. The paper concludes with related efforts to introduce computational thinking in kindergarten through 12th grades and higher education along with a discussion of insights computer science offers aerospace by revisiting questions from the introduction.**

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.I010193

## I. Introduction

AEROSPACE engineering curricula are at a crossroads. The aerospace community recognizes that today's and tomorrow's aerospace engineers will rely on computers for most of their work, ranging from analysis, to experimental data acquisition/processing, to real-time flight management. However, constituents within the community do not share a common understanding of what we need to teach young aerospace students to best prepare them for careers in industry, government, and academia. What "fundamental knowledge" in computing do 21st-century aerospace students need to acquire before graduation? Do they need basic skills in information technology (IT)? Do they need proficiency in one or more programming languages and, if so, which ones? Do they need familiarity with "software applications" for computer-aided design (CAD), finite element modeling (FEM), and/or computational fluid dynamics (CFD)? Do they need procedural or object-oriented programming, or both? Do they need principles of logic/finite-state machines or software engineering, or both?

Additionally confounding our search for answers to these questions are emerging capabilities in automatic code generation and cloud computing. Then, additional questions arise, such as "Why do aerospace students and practitioners need to understand how to program in a traditional language when we will likely autogenerate as much code as we can in the future?" and "What aspects of the complex worldwide data storage, mining, and processing network do aerospace students and practitioners need to understand versus simply use as a black box?"

Typical answers to these questions are biased by the traditional aerospace "pillar" (structures, gas dynamics, or dynamics and control) on which the aerospace answerer focuses his/her attention. The structural engineer indicates a background in CAD and FEM is essential; the gas dynamics expert believes some knowledge of CFD and its underlying numerical algorithms is essential; and today's control engineer relies on the toolboxes in MATLAB and Simulink [1–3] to model, simulate, and regulate complex physics-based systems. My first department chair† indicated he "wanted students to hug their computers" (rather than fear them) and "be able to write a few lines of code when they graduate." These are modest but important goals. Most agree that numerical methods are important to aerospace, but there is not even universal agreement on how numerical methods should be taught. Are these methods to be taught traditionally, focusing in lecture on the math underlying the numerical computations but then requesting students generate procedural implementations executing as a single thread, on a single core (i.e., with serial rather than parallel execution), using a language such as C, [procedural] C++, Fortran, or Python? Or, is it sufficient to exclusively use MATLAB so students can focus mostly on the math by relying on MATLAB's toolboxes and automatic data management that "black box" implementation details? Or, should numerical methods be seen as a series of algorithms students develop with the goal of better understanding principles in object-oriented as well as procedural program design and implementation; capitalization on the multicore processor architecture, the graphics processing unit, and effective use of distributed or cloud computing resources?

Because the AIAA community primarily comprises constituents from the traditional aerospace disciplines, the aforementioned questions purposely take the perspective of the aerospace engineer rather than the computer scientist/engineer focusing his/her career on aerospace applications. Nowhere was it more clear than serving on a Software Engineering Education panel at the 2011 AIAA Aerospace Sciences Meeting that there is a giant gap between the priorities of the traditional aerospace community and the priorities of computer scientists dedicated to solving important aerospace challenges such as those associated with the development and augmentation of robust, complex, and verifiable software to autonomously or semiautonomously manage air and space vehicles and networks. At this 2011 Software Engineering Education panel, results were presented on a large-scale AIAA study regarding structured programming education requirements for aerospace [4]. Software engineering experts were also present, focusing on issues with large-scale systems, big data, etc. Whereas both traditional AIAA and software engineering experts supported structured programming, the pedagogical goals of the two groups had only modest overlap. Still, both groups provided useful viewpoints. In the AIAA study, surveys were conducted of aerospace engineering department chairs, aerospace managers and employers, and

Presented as Paper 2013-0594 at the Aerospace Sciences Meeting, Grapevine, TX, 7–10 January 2013; received 1 October 2013; revision received 28 February 2014; accepted for publication 15 May 2014; published online 14 August 2014. Copyright © 2014 by Ella M. Atkins. Published by the American Institute of Aeronautics and Astronautics, Inc., with permission. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 2327-3097/14 and $10.00 in correspondence with the CCC.

*Associate Professor, Aerospace Engineering. Associate Fellow AIAA.

†William Fourney, available online at http://www.aero.umd.edu/faculty/fourney.

young aerospace professionals. Results indicated that most young aerospace professionals were exposed to some structured programming but that such courses were typically never connected with mainstream aerospace course content. Employers indicated education in structured programming provides foundational skills for problem-solving. In the survey, there was substantial focus on which "programming language" was favored among a list including C, C++, Fortran, Java, MATLAB, and Excel. Although the survey might have had greater pedagogical impact from the computer science/software engineering perspective if it had instead focused on fundamental topics such as recursion, abstraction, decomposition, and big data/code management, survey questions regarding programming language did reveal interesting trends. Overall, department chairs placed importance on MATLAB, but employers indicated training in "tools" such as MATLAB should not come at the expense of education in structured programming. This finding is central to a primary thesis of this paper, which is that aerospace departments have placed so much emphasis on using tools to facilitate calculus-based modeling and analysis concepts that they have lost sight of the fact that students never really learn what is "under the hood" of these tools. Consequently, as industry has indicated in this AIAA survey, recent graduates are not typically prepared to contribute to aerospace projects in which custom software development or even understanding is required.

Although it is clear that aerospace undergraduate degree requirements will never provide computing exposures comparable to those found in computer science programs, it is also clear that aerospace programs need to evolve to require additional computing or information technology content to better prepare students for 21st-century careers [5]. The primary goals of this paper are therefore to 1) promote a better understanding of the gap between computer science and traditional aerospace views of computing and associated curricula, 2) define and discuss specific challenges that must be addressed, and 3) explore different options to provide aerospace undergraduates more comprehensive and frequent exposure to computing concepts.

First, and perhaps foremost, definitions from the community are provided to shed light on what the computer science community means by terms such as "computer science," "computational thinking," and "software engineering." Next, some common misconceptions are directly addressed in an attempt to gain a solid footing for the ensuing discussion. These misconceptions are critical to address because they feed into the incorrect notion that computing is not essential to cover more fully in the required aerospace undergraduate curriculum.

There are substantial barriers to improving computing content in aerospace curricula. First, and perhaps foremost, is the credit limit. Whereas most aerospace faculty would not mind students being better exposed to computing concepts, they are not willing to "give up" content from the legacy curriculum, particularly in their individual areas of expertise. Second, there is significant debate on which computing content is essential and why, as discussed previously. Third, many aerospace academics in particular view "computing" as "writing code" and "using tools" rather than as a science in its own right, resulting in the assumption that students can "pick up programming (or tool use) on the side" rather than through required curricular content. These barriers cannot be fully overcome through a single solution that can be applied to all aerospace programs. Instead, this paper contributes by identifying barriers and proposing means by which they can be better understood, and thus gradually overcome.

The aerospace community has a long heritage of building analysis tools (codes) on an underpinning of physics-based models; thus, curricular advancements within aerospace have presumed the most important computing knowledge to share with students is related to numerical methods used to solve physics-based modeling problems, potentially at a large scale. Computations involving physics-based models are but one application of computers and not the only application relevant to aerospace engineering. This recognition is central to understanding and addressing barriers against including additional content in computer science that does not revolve around calculus-based analysis. Next, a discussion of the origins and uses of physics-based models are provided; this is not to argue for the adoption of more advanced physical models in aerospace (e.g., relativity or quantum) but to promote reflection that calculus-based physical models represent one abstraction of physical system behavior. An understanding of and comfort with discrete models (to describe objects, natural language, digital data representations) are essential for aerospace systems comprising complex networked systems that must interact with human operators and users. Logic or state-based models are essential to capture and reuse results from analyses or algorithm development, and to manage flight vehicle systems that are becoming increasingly distributed, complex, and autonomous. Example curriculum modules that simultaneously promote discrete (symbolic) and physics-based models and algorithms are presented next. The intent of delivering such modules is to not only expose students to additional data structures and algorithms but to also enable them to compare and contrast discrete and continuous modeling approaches, and see how both are useful for aerospace modeling and analysis.

To situate the discussion, a review of required computing courses in top-ranked aerospace programs is provided, followed by a more specific review of course requirements for aerospace engineering, and computer science and engineering at the University of Michigan (the author's home institution). A comparison of the content and the flexibility in these majors is provided along with a discussion of input to date from the Michigan Aerospace Industrial Advisory Board. Options for today's aerospace undergraduates to obtain expertise in software engineering and, more generally, computer science are presented, with a discussion of which options are most feasible in the near-term versus which options are recommended longer term. This paper presents examples of how and why computing must be further infused into the aerospace curriculum; a section is then included on related efforts to infuse computing into aerospace or, more generally, engineering curricula to provide additional ideas the aerospace educator can and should consider. This paper concludes by revisiting the aforementioned questions in the continuing quest to engage aerospace students in educational programs that truly reflect content in computing as well as "aerospace" science.

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.1010193

## II. Brief Summary of the History and Science of Computing

Computational devices substantially predate the modern digital computer. Mechanical devices such as the abacus and slide rule provided some assistance with mathematical computations. Babbage's 19th-century analytical engine [6][‡] proposed the arithmetic logic unit, operations such as selection and iteration, and use of integrated memory. The first digital computers were developed in the mid-20th century as a means of information processing and storage. The computer has subsequently undergone an unprecedented metamorphosis to achieve today's high-speed distributed computing networks. Today, challenges in "big data" and "cloud computing" are at the forefront, representing a fundamental shift away from focus on faster, more capable single-computer solutions [7]. Fault tolerance and cybersecurity also present substantial challenges: How do we guarantee uninterrupted and uncorrupted information handling given supercomputers or computer networks with millions, or potentially billions, of processing and communication elements?

Computing algorithms are founded in the mathematics of logic. Boolean algebra [8,9] proposed a mathematical logic from which switching algebras based on logic gates would be developed. The principles of Boolean algebra have provided a sound basis for circuit design, computational theory, and formal algorithm analysis and verification with constructs such as the binary decision diagram [10]. Whereas Boolean algebra supports the representation and analysis of a "snapshot" of a digital system at any particular time, the concept of a state machine emerged to represent evolving sequences of states. In the finite-state machine (FSM) or finite-state automaton [11], the system occupies one of a finite set of states at any one time and transitions between states as a function of observed events (incoming data). The Turing machine [12], formed by adding

[‡]Several citations to Wikipedia are intentionally included in this paper because Wikipedia text represents, in many cases, an evolving community consensus and because computing education, including judicious data acquisition and use, is quite related to the topic of this paper.

a theoretical "infinite tape" representing large-scale nonvolatile memory to the FSM, removes the finite memory limitation inherent in the FSM. The Church–Turing thesis [13], which holds that a function is algorithmically computable if (and only if) it is computable by a Turing machine, provides a basis for the notion of "computability" accepted today.

Fundamental algorithm development and analysis are still based on logic, state machines, and computability theory. However, computer scientists are challenged with new questions in algorithm and software analysis and development related to addressing the big data and cloud computing challenges previously cited [7]. Effective abstractions allow large problems to be modeled more simply. Codes continue to grow in size, must often be portable, and must be able to effectively evolve over the product's lifecycle. New applications will only be competitive if they reuse [modular] code, and large codes will require effective workflow management to debug, integrate, and manage code bases. Validation and verification, particularly with large codes and "big" datasets, require development of highly robust and potentially adaptive algorithms implemented across heterogeneous distributed computing networks.

The field of computer science has emerged to advance the theoretical and application-driven concerns associated primarily with the use of the digital computer. Computer science deals with the theoretical foundations of information and computation, together with practical techniques for the implementation and application of these foundations [14]. Its subfields can be divided into theoretical and applied disciplines. *Theoretical* computer science disciplines include information and coding theory, algorithm and data structures, programming language theory and formal methods, distributed systems, and database/information management. Applied computer science disciplines include artificial intelligence (AI), computer architectures, computer graphics and visualization, computer security and cryptography, computational science, information science, and software engineering including real-time systems.

As is the case in other disciplines, computer science is viewed from different perspectives, depending on area of focus [15]. This further confounds questions in how to introduce the "science of computing" to aerospace students. The *rationalist* paradigm, adopted by theoretical computer scientists, models software/algorithms with mathematical objects and analyzes program correctness through deductive reasoning. The *technocratic* paradigm, adopted in software engineering, models programs as data and assesses program reliability using manually or automatically generated test suites. The *scientific* paradigm, adopted in applied areas such as AI, defines computer science as a natural empirical science, considers programs in the context of mental processes, and evaluates performance through a combination of formal deduction and experimentation [15].

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.1010193

## III. Computational Thinking: A Pedagogical Perspective

The preceding summary provides a glimpse at the breadth, depth, and diversity associated with the computer science field. Before moving on, it is useful to reflect on the purpose of this paper, which is to investigate crosscutting education in aerospace and computer science. Clearly, not all aerospace students will become experts across all the theoretical and applied computer science disciplines previously listed. However, aerospace students might learn select "theory" on the foundations of computer science, e.g., logic, state machines, and data structures. Perhaps they also can learn select "applied" knowledge (e.g., from computational science and software engineering) that can assist with aerospace algorithm development, implementation, validation, and use. (Note that choice of programming language is purposely not emphasized in this paper. Most fundamentals of the theoretical and applied disciplines of computer science are independent of the specific language chosen. The choice can then be driven by conveniences such as support for development, ease of use for the particular application, and current/legacy usage across the community of practice.)

The computer science community has recognized the need for students to understand, not just use, computing devices and applications. Computer scientists, particularly those involved in kindergarten through 12the grade (K–12) and undergraduate education, also have recognized that legacy educational programs outside computer science are deficient in teaching theoretical and applied computer science basics. The term computational thinking [16] has been proposed to "represent a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use." Concepts such as selection, iteration, recursion, abstraction, and decomposition are cited as foundational. The goal is to formulate "what can be computed and how to compute it" [16].

Freshmen-level engineering computing courses typically cover procedural programming, including selection, iteration, and functional decomposition. Second-year computer science courses in discrete math and object-oriented programming/introductory data structures are not typically required for aerospace students. This means most aerospace students graduate without any formal exposure to advanced computer science concepts beyond those taught in "freshman programming." Additionally, given that many students have no prior background, clear thought in more advanced concepts such as recursion and abstraction is not possible, even when covered in a first-year course. While basic computational thinking processes may be jumpstarted by a single well-taught course, repeat exposure is also essential to ensure students continue to use and extend their knowledge. How, then, can more advanced computing concepts and repeat exposure best be offered, and in what context, to further expand aerospace students' computing knowledge?

Educators seeking improved computational thinking skills for everyone, not just aerospace engineers, have investigated the "human question," or how humans in general, not just computer scientists, can better understand computing [17]. To do this, researchers have investigated what computing concepts are "natural" versus "learned." Results summarized in [17] suggest the conditional "if" is natural, yet the "else" associated with branching in selection must be learned. Iterative repetition is also not natural. Such results suggest that production rules [18] used in AI languages such as Soar [19] may be more natural for programming than traditional languages. Similarly, experiments have suggested that object-oriented thinking is not natural, even though modularity through object-oriented programming is viewed as essential for building, managing, and reusing large code bases. So, how can aerospace students be best equipped with computational thinking skills? Certainly, aerospace students will be somewhat prepared to think using concepts from algebra- and calculus-based mathematics given their background, whether natural or learned. Perhaps education in computational thinking for aerospace students can effectively build on this combination of learned mathematical background and natural production-rule-style thinking. The primary challenge, then, is to define which of the foundations in computational thinking is most important for aerospace students to learn.

Contemporary aerospace educators have cited numerical methods courses as a means to combine learned mathematical principles with natural (and learned) computational thinking processes. Whereas such courses could theoretically be designed to further principles in computational thinking using numerical methods relevant for aerospace as the motivating application, most existing courses in aerospace do not serve this purpose. Instead, they focus on specific means by which difficult calculus-based problems can be solved for aerospace or, more generally, physics-based applications. As an example, consider courses such as "Computational Methods in Aerospace Engineering" [20] and "Introduction to Numerical Simulation" [21] offered through Massachusetts Institute of Technology's (MIT's) OpenCourseWare system. Students program in MATLAB for assignments, but typical lectures focus strictly on the mathematics for integrating/solving differential equations, solving nonlinear systems, and probabilistic simulation. To be clear, such coursework is quite valuable for many aerospace students. Such courses, however, are valuable because of the mathematical methods they introduce, and not because of their careful coverage of fundamental computing principles. [Instructors of numerical methods courses often introduce important computing concepts, such as computational complexity, numerical precision (as a specific subclass of data representation), and distribution of algorithms across parallel processing resources. Such courses therefore have the

potential to provide essential computer science content to aerospace students once the computational thinking content offered in such a course is clearly cited and placed in the context of the broader set of computer science principles. Without this clarity, numerical methods courses can, at best, introduce computational thinking concepts in an ad hoc fashion. Students may then be able to use introduced methods but will be less likely to be prepared to innovate with respect to the computing concepts that were "swept under the carpet".] The assumption in such a course is that either the student already possesses mature computational thinking skills or the aerospace student does not require sophisticated computational thinking skills because he/she can always "abstract" the math models away from the underlying logic, data structures, and computational devices on which the numerical methods are actually implemented. Neither of these assumptions is correct. Computational thinking is not yet emphasized in K–12, and most aerospace programs have, at best, one prerequisite course in which algorithm/computational thinking is the focus.

## IV. Computer Science Meets Aerospace: Addressing the Misconceptions

Because computer science is a relatively new discipline, in that the digital computer was only available for practical use starting in the latter half of the 20th century, there is not yet a universal understanding of terminology associated with the field. Academic communities are also struggling to understand what content is important to teach versus assuming students can either pick up independently, learn later "on the job," or never learn because it is not important or fundamental. Some common misconceptions the author has encountered are discussed next.

### A. Computational Thinking Skills Do Not Equal Information Technology Skills

Twenty-first-century engineering students typically enter undergraduate programs quite familiar with computer-based collaboration tools, including email and chat as well as office applications. Most have extensively played video games during K–12 and are more proficient with cell phone and tablet texting than older adults could ever hope to become. These so-called "millennials" or "net geners" have proficiency with search engine use and find it intuitive, and even fun, to identify, download, and rapidly scan digital documents. They value education and are voracious consumers of information, yet on average, they will have spent far fewer hours reading books (hardcopy or online) than texting or emailing, and they are likely to read only a part of each digital resource unless they find it particularly engaging [22–24]. There are some who believe that students' minds are changing, migrating toward increased impatience but improved ability to multitask and collaborate with others throughout the learning process. Students anticipate a collective, distributed intelligence they can always access, but they are challenged by a pervasive need to judge which information sources are accurate [25].

Because of these net-gener skills, 21st-century high school and college students are considered good with information technology, where IT "is broadly considered to encompass the use of computers and telecommunications equipment to store, retrieve, transmit and manipulate data" [26]. IT skills are useful in engineering and in life, and educators are working to understand how to best engage the multitude of information sources available to today's students within and outside of the classroom. Whereas questions in "how to best use IT" can inform all pedagogical efforts, these questions do not focus on understanding what is under the hood of computing devices with respect to the underlying design and operation of the hardware and software. From the computer science perspective, entering college students that possess only IT skills can only be users and operators of computing devices, much as they are users and (recently) drivers of automobiles.

It is critical to understand and appreciate that skill in IT does not imply a background in computer science or a strong ability to think computationally or solve problems using efficient data representations and algorithms. In fact, the 21st-century student who has spent so much of his/her life treating computing devices as black boxes for information access and communication may in fact feel quite threatened by activities or classes that require going inside the black box. The learned expectation of rapid information access and processing can hinder students in appreciating the educational value of exposures to computer science education, including gaining experience with low-level algorithm and software development.

### B. Computer Science Does Not Equal Computational Science

Computational science, as previously stated, is one of the applied subdisciplines of computer science. Computational science or scientific computing "is concerned with constructing mathematical models and quantitative analysis techniques and using computers to analyze and solve scientific problems" [27].[§] Numerical methods fall under the classification of computational science. Experts in computational science focus their attention on a focused subset of computer science. Floating point data representations and their use in numerical computation algorithms are emphasized. Floating point data evolve over time, but data structures typically do not. Computations may be distributed and tend to be costly due to the size of the model and complexity of the numerical algorithm. This is in contrast to complexity that also arises from heterogeneity and dynamicism in information content, processing/communication demands, and resource availability. To compare and contrast in aerospace contexts, consider multidisciplinary design optimization (MDO) for aircraft versus systemwide information management (SWIM) [28] for the next-generation air traffic management system. MDO is at the core of computational science. By contrast, SWIM relies on system-of-systems models comprising discrete (logic) as well as continuous data. SWIM relies on hierarchical organization of abstract datasets to manage complexity in communication, as well as processing and memory requirements. Air and space systems also face computer science challenges in cybersecurity, software engineering, and autonomy (from the AI perspective) not considered in computational science.

### C. Implementation Is Not a Dirty Word

The scientist seeks to better understand natural phenomena by building and organizing relevant knowledge in the form of testable explanations and predictions [29]. The engineer, on the other hand, seeks to apply scientific principles to design, develop, construct, and/or operate devices in a manner that can be understood and predicted with respect to performance, lifetime, safety, etc. [30]. The "devices" of aerospace engineering may be flight vehicles, flight vehicle subsystems or components, or systems associated with individual vehicles or vehicle groups. In computer science, an "implementation" is a realization of a technical specification or algorithm as a program, software component, or other computer system through programming and deployment [31]. Implementation is the step following the design and development of an algorithm, analogous to the "construction" process associated with engineering hardware devices. "Operation" of modern aerospace systems requires that both their hardware [devices] and requisite software [implementations] function. In recent years, educators and practitioners have collectively concluded there is value in offering substantive design, build, fly or alternatively conceive, design, implement, operate experiences to aerospace engineering students. Implementation is a key to understanding how a theory or algorithm will actually function in practice. It is therefore central to aerospace education.

Unfortunately, academics focused on the traditional aerospace pillars typically do not view the implementation of an algorithm as central to education or their research. Evidence of this attitude is clear as we witness an almost universal embracement of tools such as MATLAB in traditional undergraduate and graduate aerospace courses. Such Matlab tools black box data representation and algorithm logic leaving students

[§]The subheading of the Wikipedia page on computational science states "not to be confused with computer science."

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.I010193

will little to no background in these fundamental concepts. Additional evidence of this attitude is found by comparing and contrasting the content of articles found in the literature. For example, aerospace control theorists relying on simulations at least in part to validate their work carefully present mathematical models and formally analyze them, but few such publications provide careful descriptions of simulation (or hardware) implementation specifics and how they might impact results with respect to computational aspects such as data representation and precision, limitations of black-box tools [e.g., ordinary differential equation (ODE) solvers], effects of concurrent potentially asynchronous execution when multiple threads or processes may be involved, modeling and careful consideration of time delays (based on realistic processing and communication models), etc.

### D. Computer Science Education, Including Programming, Is Relevant Despite Automatic Code Generation

The author grows increasingly tired of hearing how current and future generations of aerospace students will not really need to know how to program, and that instead, code will simply be automatically generated. This argument has been presented as a means of advocating applications such as MATLAB and Simulink that hide data representation and management details, and that provide a wealth of prebuilt toolbox functions that minimize overhead due to lower-level coding. Although the author is a strong advocate for choosing the best language for a particular application, the "auto-code-generation" argument misses the mark with respect to education. Additionally, it is still the case that, to date, most automatically generated code is neither as efficient nor as readable as manually written code; this includes MATLAB-generated C++ code the author generated within a week of submitting this paper.

Consider a "calculator analogy." Despite the proliferation of extremely low-cost calculators, elementary (kindergarten through fifth grade) educators still require that students learn how to add, subtract, multiply, and divide. In fact, students spend years mastering these basic math skills. Why spend so much time mastering basic math if later all that will be required is to press the correct buttons on a calculator or to enter the correct commands into a spreadsheet application? Most would argue that basic math is fundamental to our ability to think about and solve quantitative problems, particularly those requiring innovation. I agree. It is therefore baffling to the author that many mathematically capable, aerospace engineers advocate for black-box code generation rather than improved education in computational thinking, including algorithm design and implementation.

As a result of a strong and exclusive dependence on MATLAB and Simulink for "code development" at the undergraduate and graduate levels, and even in industry, coding has certainly become easy and less tedious, as is basic math when given a calculator rather than pencil and paper. The result is that today's aerospace students graduate without learning about memory management or even simple data structures, such as the linked list vital to efficiently represent dynamically evolving entities such as sparsely connected graphs. Instead, students insist on representing graphs in the manner suggested by MATLAB: as a matrix of edges that must be copied to new memory when the matrix is "grown" (a node is added) and that often contains a large number of unneeded zeros (representing no connection between a pair of nodes). Let us suppose a student writes a complex control law for a networked system with changing connectivity represented in a graph. In MATLAB, matrices will be used to represent this graph, and the requisite embedded control law can be easily debugged in MATLAB/Simulink on a powerful desktop computer or server without too much overhead or delay. Then, when code is then autogenerated for the embedded system, matrices will be used in the generated code whether such structures are efficient or not because the automatic code generator is simply translating the user's code and data structures to a lower-level language. A complete rewrite of the code by an embedded programmer who understands data structures will then be required if the generated code cannot be executed in real time on the memory-limited speed-constrained embedded processor. This scenario happens all the time in industry. Educating aerospace students in even the basics of data structures and memory management will avoid the all-too-frequent recurrence of this costly and time-consuming scenario.

### E. Your Cyberphysical System May Not Be My Cyberphysical System

The emerging field of cyberphysical systems (CPS) [32] focuses on modeling and control of computing and vehicle (mobile) systems operating in and making decisions about a complex physical environment. However, the nature of data, translation, and abstraction of this data, and subsequent control decisions made in CPS, depend substantially on both the application and the perspective taken by the researcher studying CPS [33]. For example, CPS have been proposed for domains ranging from medical monitoring [34] to automotive (engine control) applications [35]. CPS decisions range from coordination [36] and fault-tolerant control [37] of physical actuation decisions to ensuring the security of information [38] acquired from the network/cloud. In summary, CPS are inherently multidisciplinary, challenging educators and practitioners to maintain knowledge of fundamental and application concepts that cut across, at a minimum, the fields of dynamics and control (control theory) and computer science (real-time computing). Next are summaries of the perspectives to CPS taken by each of these two groups. Whereas there are many common aspects to the CPS that cut across all perspectives, the emphases taken in modeling and decision-making still tend to be quite discipline specific, leading to further challenges in developing a CPS curriculum that is sufficiently perspective independent to enable those educated in CPS from one perspective to successfully interact with those educated with respect to another viewpoint. To date, most CPS fundamental research has origins in either the dynamics and control community or the real-time systems branch of computer science. Interpretations by each are discussed next.

Historically, models used by the dynamics and control community have been specified with continuous-time differential equations. For mobile vehicles, these models typically specify the rigid-body motion of a body coordinate frame with respect to an inertial reference frame. Robotic or flexible-structure vehicles may require additional modeling elements to account for nonrigid degrees of freedom. The control theory community has begun adapting discrete modeling and computational methods originally derived for purposes outside control theory, e.g., finite-state automata from computer science and neural networks from biology. However, the goals of control-oriented research programs remain tied to designing physical system controllers, where a system can be interpreted as one vehicle or a group of networked vehicles. The focus when modeling discrete control systems as CPS, therefore, has been on verifying properties such as stability and controllability rather than on representing, manipulating, and communicating potentially large, dynamic data structures.

The field of real-time computing [39] is much newer than are dynamics and control, given that computers were only scaled to a size that could be embedded in physically mobile systems a few decades ago. Computing technology and software engineering have, however, progressed extremely rapidly, as have communication networks. Real-time computing (RTC), or reactive computing, is the study of hardware and software systems that are subject to "real-time constraints," e.g., response times measured from the initial event (task arrival) to system response (task completion) [40]. Computing theory forms the basis for the formal specification and characterization of static and dynamic data structures, computing algorithms, and communication protocols to which real-time computing analyses are applied. In large-scale networks, the set of computing and communication tasks to schedule, many with precedence and exclusion constraints, can be quite large. Large-scale scheduling applications have therefore encouraged RTC researchers to develop approximate algorithms and hierarchical network structures that are scalable. The RTC student will learn models of computation, including data structures, algorithms, and complexity analysis. In the context of the CPS, the RTC student will therefore define problems in terms of data to be packaged, transferred, processed, and stored, and the algorithms by which the associated computing and communication tasks are achieved.

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.I010193

**Table 1 Required computing courses at top-ranked aerospace departments**

| Rank | University | No. of required computing courses | Description |
|---|---|---|---|
| 1 | Massachusetts Institute of Technology (starting class of 2016) | 1[a] | Introduction to computers and engineering problem solving (Java) |
| 2 | Georgia Institute of Technology | 1 | Computing for engineers (MATLAB) |
| 3 | University of Michigan | 1 | Introduction to computers and programming (C++ and MATLAB) |
| 4 | Stanford University (mechanical engineering) | 1 | Programming methodology (Java) |
| 5 | University of Illinois | 1 | Introduction to matrix theory (MATLAB/mathematics-centric) |
| 6 | Purdue University | 1 | Programming applications for engineers (C and MATLAB) |
| 7 | California Institute of Technology (mechanical engineering major; aerospace minor) | 1 | Introduction to computer programming (Python) |
| 8 | University of Texas at Austin | 2 | Introduction to computer programming; engineering computation (numerical methods) |
| 9 | Princeton University | 2 | Introduction to computer science (C); microprocessors for measurement and control |
| 10 | University of Maryland in College Park | 2 | Aerospace computing (C++ and MATLAB); flight software systems |

[a]MIT requires all aerospace/astronautics students to take only one common computing course, but numerous technical electives enable undergraduates to place significant emphasis on computing in their coursework without the need to add credits beyond those required for the degree should they choose this option.

This most common RTC interpretation of CPS is therefore quite distinct from the control community's interpretation of CPS. This disconnect results in substantial differences in each community's views of what is "essential" for education of even the emerging CPS "expert."

## V. Existing Undergraduate Curricula and Feedback from the Aerospace Industry

Before discussing how computational thinking and computer science education for aerospace might be improved, we first ground the discussion by reviewing required undergraduate computing coursework. First, we examine computing courses currently required for top-ranked aerospace engineering undergraduate degree programs. Next, we examine more closely the curricular requirements for aerospace engineering at the University of Michigan, which is the author's current institution. For comparison purposes, the University of Michigan curriculum in computer science and engineering is also summarized.

### A. Top-Ranked University Aerospace Engineering Computing Curricula

The AIAA study on structured programming education requirements for aerospace [4] concluded that recent graduates are somewhat deficient in their knowledge of structured programming. This study further concluded that one required structured programming course, given appropriate content and high-quality teaching, might address this problem adequately. Could the solution be so simple? To further investigate, the author compiled a list of the top 10 aerospace engineering undergraduate programs in accordance with the *U.S. News and World Report*'s 2013 rankings[¶] and then examined computing course requirements for each school's undergraduate aerospace program. (Some universities have degree programs in aeronautics and astronautics, aerospace and mechanical, etc. Each degree program referenced by the *U.S. News and World Report*'s rankings was examined for this study.) Although many courses might use computer applications (e.g., building a CAD model, running a simulation, or plotting laboratory data in MATLAB), only required courses with significant focus on structured programming and other computer science subdisciplines were classified as computing.

Table 1 shows a summary of computing courses required at the top 10 aerospace engineering schools, beginning with the number-one-ranked school (Massachusetts Institute of Technology) and ending with the number-10-ranked school (University of Maryland in College Park). Data on number of computing courses required for each aerospace undergraduate program, as well as course titles and programming languages covered, are provided for each institution. The first and most obvious conclusion from these data is that all top-10 aerospace programs require either one or two courses that emphasize computing. Further, as shown in the table descriptions, there is significant variation in content between the first computing course required at each university. Most of the introductory computing courses introduce structured programming in some form, with the exception of the University of Illinois course that focuses on matrix mathematics with support from MATLAB. Between just these 10 schools, five different languages are used: C, C++, Java, Python, and MATLAB. Some of these languages can support object-oriented programming (C++, Java, and Python); others are more appropriate for procedural programming (MATLAB and C). Three universities require two computing courses. Interestingly, the University of Texas at Austin recently concluded a period in which they dropped the introductory computing course from the curriculum, requiring only the one engineering computation (numerical methods) course. The University of Texas at Austin (UT-Austin) has reinstated their "Introduction to Computer Programming" course for 2013–2014 and beyond because they learned that introductory programming content was never successfully included in the numerical methods course, and because their External Advisory Committee indicated that the UT-Austin curriculum "needed more programming instruction" [41].

Two overall outcomes can be deduced from Table 1. First, recent aerospace graduates will enter the workplace with varied computing knowledge, both with respect to programming languages, code design, and implementation technique (procedural versus object-oriented). Second, many aerospace students will graduate with exactly a one semester of exposure to fundamental computing concepts, particularly at institutions (like the University of Michigan) where the majority of the aerospace faculty do not advise students to supplement required content with computing electives. The AIAA study conclusions that aerospace undergraduate structured programming education is deficient (including at top-10 schools) and that one structured programming course may be sufficient are together inconsistent with the fact that all top-10 aerospace departments already require at least one structured programming course [4]. It is highly unlikely that most of these 10 schools teach their required structured programming course ineptly. It is more likely that at least one AIAA study outcome will need to be updated: that, indeed, multiple computing course exposures and/or serious computing concept infusion into more traditional coursework are instead required to prepare graduates adequately.

[¶]Data available online at http://colleges.usnews.rankingsandreviews.com/best-colleges/rankings/engineering-doctorate-aerospace-aeronautical-astronautical [retrieved February 2014].

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.1010193

### B. Aerospace Engineering at the University of Michigan

At the University of Michigan, all College of Engineering students have the same requirements in their first year, enabling students to declare their major at the end of their second first-year term. In addition to humanities and social sciences, all University of Michigan engineering students take four math courses (Calculus I through Differential Equations (Calc I–Diff Eq)), two physics courses with laboratories, one chemistry course with laboratory, and two collegewide engineering courses: one introducing students to project-based engineering [titled Engineering 100 (Engr100)] and one four-credit introduction to computers and programming course [titled Engineering 101 (Engr101)]. (An accelerated introduction to programming option, titled Engineering 151 (Engr151), was introduced by the author to help students with a programming background find a more appropriate challenge. Few students electing Engr151 pursue aerospace however.) Whereas Engr101, taught multiple times by the author, introduces freshmen to computational thinking in the context of C++ and MATLAB, it is an "island," i.e., most students enter with no programming background and then take no further courses in this area. Students who do not major in electrical engineering and computer science see little further computational thinking content in their undergraduate coursework. Engr101 therefore has no reinforcement in most majors, and thus has limited impact on a student's preparedness for more advanced computing concepts.

The required aerospace-specific undergraduate curriculum at the University of Michigan is structured around three pillars: structures, gas dynamics, and dynamics and control. Students are not offered tracks: many faculty contend that students are not yet prepared to decide what aerospace path they want to take, thus a unified curriculum focusing on the fundamentals is most appropriate. Each of the three pillars has a sequence of required courses that build to senior capstone design and laboratory courses. In structures, students take materials science and a sequence of two structures courses. In gas dynamics, students take a series of two courses in aerodynamics plus a propulsion course. In dynamics and control, students take a sophomore dynamics course plus an introductory aerospace course that covers atmosphere models, coordinate systems and kinematics, steady flight, and systems theory. Students then take a space flight mechanics and aircraft dynamics and control course sequence. A sequence of two laboratory courses, a seminar course, and a capstone design course (aircraft and space systems options are available) round out the required course list. In terms of software, MATLAB and Simulink are the most widely used software packages. Their use, however, focuses almost exclusively on promoting understanding of concepts related to calculus-based modeling and simulation, using built-in functions that black box most all aspects of how data are represented, processed, and communicated.

Two aerospace courses have recently shown some promise to expand student exposure to computational thinking. The introductory aerospace course providing coverage of steady flight principles has evolved to introduce students to fundamentals of logic and state machines. Another course, to be first required in 2014, will provide an early design–build–test experience for undergraduate sophomores. Although primarily serving the need for more hands-on education in the curriculum, this course will provide some embedded programming experience for students, including data acquisition and real-time processing and communication with limited resources. To some extent, these two courses will better prepare aerospace sophomores for a more in-depth education in computer science. The challenge then is to follow up with significant computer science exposures in the junior, senior, and graduate years without continuing to require that students with strong interest in this broad area double major in computer science. Note that, while the two sophomore aerospace classes introducing logic and introductory embedded programming are vital computational thinking exposures, they do not introduce concepts such as object-oriented programming and data structures. This gap makes it necessary for aerospace students interested in enrolling in upper-level computer science courses to take an extra sophomore-level computer science course covering these concepts or to gain sufficient background through extracurricular activities and/or online tutorials/courses.

### C. Computer Science and Engineering at the University of Michigan

In computer science and engineering (CSE), undergraduate sophomores take discrete mathematics and statistics courses. A sequence of two courses introduces sophomores to object-oriented programming, data structures, and algorithms. Foundations of computer science and computer architecture courses, plus a course in technical communication, round out the program core requirements. CSE offers more options or tracks to its undergraduates than aerospace, which follows a single track, except for the senior capstone course. CSE students are required to take a major design experience course from an approved list. Students also must take 26 technical elective credits, of which 16 must be upper-level CSE courses. Students can choose from a wide variety of upper-level courses ranging from artificial intelligence and compiler construction to operating systems, formal verification, robotics, and user interfaces.

The flexibility in the CSE program provides an opportunity for students interested in CPS to pursue a related course sequence within the major, enabling broad or deep coverage in related topics such as real-time computing. Although technical electives offer CSE students the option of broadening their CPS-related expertise to control theory or more generally calculus-based modeling courses, upper-level students who are not double majoring and who have focused most of their attention on discrete models, algorithms, and data structures choose not to pursue the more advanced courses in physics-based modeling.

### D. Feedback from the Aerospace Industry

Academic units typically invite industrial advisory boards to attend annual program reviews and provide recommendations for the department. Feedback is broad but often focused on curriculum issues, since industry has a strong interest in recruiting graduates who are suitable for employment at their companies. The author has some experience with advisory board feedback at two universities. Each year, the advisory board discusses the need for improved computing education. The problem is that, for aerospace, most board members have little personal expertise in what has been defined here as CPS, so most think of computing in terms relevant to their aerospace subdiscipline. For example, structures experts consider computer-aided design, finite element modeling, and computer-aided manufacturing software packages to be critical exposures for undergraduate students. Although these packages provide significant insight into the design and manufacturing of aerospace structures, they do not promote the kind of fundamental understanding of computational thinking [28] essential to prepare students for further study of CPS.

Some members of industrial advisory boards, particularly those who have managed aircraft or spacecraft programs that have faced significant software and avionics cost overruns, have expressed support for further student exposure to software development and testing. Of particular interest to the advisory board, and to some students, is exposure to embedded system programming and/or software engineering. The challenge then is to fit such coursework into the aerospace curriculum. During the first decade of her time in academia, the author idealistically embraced support from the advisory board, believing such support would lead to curriculum change. Unfortunately, the realities of credit limits and lack of recognition by other faculty of the importance of computational thinking (much less CPS) in the required curriculum have resulted in few changes. In fact, there has been some decrease in coverage of software engineering and embedded systems concepts in some aerospace programs due to black-box applications and off-the-shelf products, such as the strong dependence on MATLAB toolboxes previously discussed. Such convenient black boxing can reduce motivation for students (and in many cases, even faculty instructors) to ensure students are challenged to learn about even basic computing or RTC concepts.

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.I010193

```
int bsearch(int A[], int length,
    int value)
{
   int low=0, high=length-1 , mid;
   while (low <= high) {
     mid = (low + high) / 2;
     if (A[mid] > value)
        high=mid - 1;
     else if (A[mid] < value)
        low=mid + 1;
     else
        return mid;
   }
    return -1;    // value not found
}
```

a. Binary Search

```
double bisect(double xmin, double xmax,
   double tolerance, double f(double))
{
   double midPoint;
   assert((f(xmin) * f(xmax)) <= 0.0);
   while ((xmax-xmin) > tolerance) {
       midPoint = (xmin + xmax) / 2;
       if (f(xmin) * f(midPoint) <=0.0)
         xmax = midPoint;
       else
         xmin = midPoint;
   }
   return (0.5*(xmin+xmax));
}
```

b. Bisection

**Fig. 1 Search algorithms (sample C++ iterative implementations).**

## VI. Analogs Between Physical and Digital Systems

In K–12 and undergraduate curricula, aerospace students are carefully exposed to progressively more challenging and sophisticated models of physics-based phenomena. Because digital computing processes cannot be described by simple continuous-valued differential equations, logic and state machines do not directly follow from education in physics. However, as computers are used to analyze physical processes, and as steps through datasets appear infinitesimal with large dataset sizes, opportunities arise to compare the two. This section describes two example opportunities to expose students to "dual thinking" across discrete computing and continuous physical systems. The first is a pair of simple algorithms for search, whereas the second describes the evolution of a dynamic systems model adopted by the control theory community to formally model switched control systems. Such exposures are win–win across disciplines: aerospace students who must use computers to solve physics-based problems gain a better understanding of discrete information representation and processing, and computer science students, who would typically focus on purely discrete models with only a vague recollection of their early calculus-based math courses, will be reminded of how algorithms can be used in real physical as well as informationcentric applications.

### A. Introduction to Search

Efficient algorithms have been developed to search through ordered data, or over a continuous-valued function. Figure 1 shows example C/C++ implementations of binary search [42] and bisection [43]. The former is typically introduced in introductory programming classes, whereas the latter is typically introduced in introductory numerical methods courses. Both have $O(\log_2(n))$ computational complexity, where $n$ for binary search represents the number of elements in (or length of) the list/array, and $n$ for bisection represents the number of intervals $(x_{\max} - x_{\min})/(\text{tolerance})$ within $[\,x_{\min} \quad x_{\max}\,]$ for continuous-valued function $f(x)$. These algorithms, especially when introduced in tandem, illustrate several important computational thinking concepts. First and foremost for this illustration is that simply changing the data representation, e.g., from an ordered list to a continuous-valued function, can provide a powerful means of repurposing even a simple algorithm such as search. Second, these algorithms can be straightforwardly transformed between iterative and recursive formulations, providing a simple means of introducing recursion, which is a more challenging concept to master than iteration. Third, both are useful algorithms that can be easily embedded within more sophisticated algorithms with low computational overhead.

### B. Evolution of Finite-State Machines to Timed Automata to Hybrid Systems

As summarized previously, finite-state machines were initially extended by Turing [12] to provide a fully general model of computability for digital computer systems. Because the FSM is defined over a finite set of alphabet symbols, however, as originally formulated, the FSM cannot represent continuous-valued quantities unless these quantities were abstracted into discrete value sets. As researchers applied computing models to embedded systems, extensions were proposed. Alur and Dill proposed timed automata [44], an extension to FSMs in which transitions were augmented with continuous-valued "timers" that could impose real-time constraints on state transition events. The timed automaton was further extended into the hybrid automaton [45], which annotated each state in a FSM with potentially unique continuous-state dynamics along with invariant conditions that must be satisfied for remaining in that state. When the invariants were no longer satisfied, the system transitioned or "jumped" to another state. Although not a topic for young undergraduates, the introduction of hybrid systems is appropriate for senior undergraduates or graduate students, and it is rich in its combination of physical and discrete system dynamics. As such, it provides a stimulating interdisciplinary modeling and analysis capability and is useful for a variety of practical aerospace systems (e.g., gain-scheduled autopilots) that transition or switch between a suite of operating modes or states. Researchers continue to pursue research in hybrid systems, both in novel applications and in improved methods for building and characterizing switching dynamics and system stability.

## VII. Crosscutting Education Examples

As discussed previously, aerospace undergraduate programs are classically organized around three disciplines: gas dynamics, structures, and dynamics and control. Courses in each area offer opportunities for infusing computational thinking education. For example, structures and capstone design courses rely on computer-aided design and finite element modeling. Each offers excellent examples of data abstraction, recursion, and hierarchical organization as well as exposure to computational science. For example, consider an assembly in a solid model. Each top-level assembly is comprised of subassemblies, which are in turn composed of lower-level subassemblies and primitive components. Assemblies are defined by data structures linking to subassemblies and components. Components are represented by data structures describing part geometry with pertinent dimensions. Solid modeling applications use assemblies to facilitate part definition and reuse, analogous in many ways to modular object-oriented software design. For solid modeling, an assembly can be represented as a rooted tree data structure. Traversing the tree, e.g., to draw a full assembly on the screen, is a recursive process through the top-down hierarchy of tree nodes. Although a static assembly can be modeled using matrix/vector representations, adding or deleting tree nodes is greatly facilitated by a more dynamic data structure type, e.g., linked lists

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.1010193

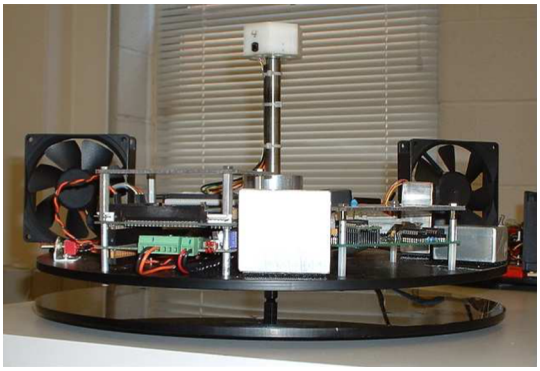

**Fig. 2 TableSat embedded system.**

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.I010193

pointing to components or subassemblies at each tree node. Aerospace courses in which students are exposed to solid modeling could therefore encourage computational thinking about hierarchical, abstract, and dynamic data representations, and about recursive algorithms to efficiently find and process data. Ensuring exposure to these concepts requires support by course instructors, as most young aerospace students will not automatically gain these insights without specific elucidation of data representation and processing concepts in lecture, as well as in homework assignments.

As another example, consider a course in feedback control where students are taught how to use real-time state estimates to accurately and stably track a reference state (or trajectory) despite uncertainties in the plant model and input/output disturbances. Control courses often use MATLAB and Simulink to simulate open-loop and closed-loop systems, and to evaluate the system through techniques such as root locus or Nyquist. In most courses, student codes are a few lines long to focus attention on feedback control rather than coding concepts. Students can be better encouraged to think computationally along at least two dimensions. First, with the advent of low-cost microprocessor and robotic systems, it is increasingly feasible for students to control a real physical system with embedded software, offering experience with real-time embedded software development and testing in addition to exposing students to the nonideal behaviors of a hardware-based (real) feedback control system. As an example, consider the single-degree-of-freedom TableSat (tabletop satellite) system shown in Fig. 2, which has been used by multiple faculty in multiple departments (computer science and aerospace) at multiple institutions (University of Maryland, Oregon State University, University of Michigan, U.S. Naval Academy, NASA Goddard Space Flight Center, and others) for embedded control and software education. If hardware is not an option, students can also be required to implement more complex code in MATLAB. For example, students might manually write a function to solve ODEs rather than using a built-in ODE solver, or students might design and implement algorithms for optimizing gains, injecting noise, and automatically assessing controller performance.

The reader may wonder why the aforementioned suggestions are somehow "new" given that computers have been used in CAD and control system courses for decades. The aforementioned suggestions are not new. What is new is dealing head on with the fact that "apps" or "toolbox functions," such as `ode45` and `lsim` in MATLAB, can easily be used while not actually understood. While aerospace students of the 21st century are adept users of numerous computing devices, it is imperative that the educational process does not "dumb down" computational thinking education relative to a past time when students were forced to write and debug low-level code because implementations of most algorithms were not so readily at their fingertips.

## VIII. Options for Extending the Aerospace Student's Computing Exposures

Although recommended highly by the author, it is unrealistic to think that aerospace programs with a well-established curriculum will quickly or substantially alter the required curriculum to accommodate new material in computing or CPS. There are therefore four means by which aerospace student exposure to computing fundamentals can be at least modestly improved:

1) Students can either double major in aerospace and CSE or major in one with a minor in the other.
2) Existing required courses can be altered or supplemented with material related to computer science.
3) Technical electives covering computing can be promoted to students.
4) One, or maybe even two, new required courses with substantive computer science and/or software engineering content can be introduced.

Option 1 is viable for top students who can handle a heavy technical course load each term or who are able to enroll for extra terms. When feasible, this option is recommended by the author as the clear path now available to gain substantial expertise in the fundamentals of both aerospace and computing. It is the author's observation that students with substantive background (and good grade point averages) in aerospace and CSE are in high demand, both in industry and for graduate programs. The author is still working on achieving significant impact with the third option listed. At the University of Michigan, an aerospace technical elective in "flight software systems" has been repeatedly offered by the author and others, but with low enrollments. Quite simply, because the CAD course is so popular and few technical elective courses are required for Michigan aerospace undergraduates, few students opt for the more computer science-based course. Some University of Michigan aerospace faculty are still scheming about ways to achieve option 4. To date, the aerospace faculty have not as a group been supportive of eliminating any existing required course credits in favor of any new more computing-related course requirements, so progress toward option 4 is stalled. Instead, instructors such as those providing the sophomore-level exposures to logic and embedded programming have pursued option 2, supplementing courses that already have significant other content with some coverage of computing principles. More is clearly needed, as we cannot just expect all aerospace students to double major (option 1) nor can we expect to fit a deep sense of computational thinking into existing courses already overflowing with required technical content. Patience, persistence, and collaboration with others supportive of increased computational "literacy" appear to be the only options available at this juncture. Wheras faculty are "in the trenches" with respect to effecting curricular change, those in the industry are highly encouraged to voice concerns and needs to aerospace departments with the hope that their collective voices will eventually be heard.

## IX. Related Work in Improving Computing Education

The aerospace, computer science, and more generally engineering communities have struggled to understand how to best expose students to the fundamental concepts associated with computing. This question is indeed raised much earlier than college. What computer science or

computational thinking can we teach K–12 students, how can this material most effectively be delivered, and what new material can be fit into at least middle and high school curricula? Educators tend to agree that motivation is a key to effective learning, with computer games and intuitive programming languages providing a fertile environment for computational thinking [46–48]. Much of the research for K–12 education is also applicable for young engineering undergraduates; both groups tend to be motivated to learn programming through embedded systems [49]. General methods for teaching college computing have been proposed, using languages such as Python to minimize the challenges in learning language syntax [50]. Educators have documented the need for computational thinking and built specific lists of computational thinking skills, and they look for opportunities to integrate computer science concepts into the disciplinary engineering curricula [51]. Vergara et al. [51] stated "Computation for engineering cannot simply be addressed with one or two courses in computing, but must be integrated as part of an engineer's training to become a Holistic Engineer." Although Vergara et al.'s hypothesis is most likely valid, and is certainly shared by this author, a central challenge as discussed previously is to convince course instructors rooted in the traditional pillars that they should modify their course syllabi to increase coverage of computing concepts while inevitably sacrificing at least a minor amount of existing content.

As shown in Table 1, all top-ranked aerospace engineering programs now require at least one programming course. If we could assume entering engineering freshmen have strong backgrounds in computing from K–12, perhaps one required structured programming course, as was recommended by the AIAA survey [4], would be sufficient, as this one course could then become an "advanced" course able to extensively cover concepts such as recursion, abstraction and object-oriented programming, and software engineering in depth. The disconnect, however, is that entering college students typically still begin from "square one" in their first college computing/programming course, resulting in the observed situation where aerospace curricula do indeed require one structured programming course; but since this is the only formal computing course aerospace graduates ever see, they neither obtain nor retain a level of sophistication in structured programming considered adequate by industry.

Aerospace educators have proposed specific options for upper-level computing course requirements. Two types of courses have been added to aerospace curricula as follow-on content to a first structured programming course: software engineering and embedded systems. Although most programs also offer numerical methods courses, these have begun to rely on black-boxed functions in MATLAB and Simulink, as discussed previously. Long [5] has not only published on the need for software engineering education, he has also successfully convinced Pennsylvania State University to require an upper-level software engineering course for all aerospace students that has been prototyped and adopted [52]. IEEE has published a guideline for software engineering education [53]; although recommended curriculum content is far too extensive for a single course, it provides a good point of reference. The author has developed a course called Flight Software Systems, which became a second required structured programming course for all University of Maryland aerospace undergraduates (see Table 1); it is offered as a technical elective at the University of Michigan. The flight software course exposes students to object-oriented programming, introductory computing theory (logic and automata), and embedded system programming for a robotic platform [54,55].

Not everyone agrees that more computing is essential in aerospace. For example, MIT no longer offers its "Aerospace with Information Technology" degree. Two decades ago [56], even MIT aeronautics/astronautics faculty who were ultimately strong advocates of bolstering faculty and student expertise in software engineering and information and decision systems expressed that "programming skills as such will not be required to graduate, but we strongly urge that students acquire them" [56]. Subsequently, at least one required course in software engineering was introduced that formally taught structured programming to aeronautics/astronautics students [57], with a variety of technical elective computing courses, such as those listed in Table 1, introduced.

## X. Conclusions

This paper has examined challenges facing improved interdisciplinary education across the computer and aerospace sciences, and it has examined current curricula as well as possible paths to infuse additional computing content into the aerospace curriculum. To provide context, the history and subdisciplines of modern computer science were summarized, followed by discussions of education in computational thinking and of misconceptions the author has frequently encountered. Curricula in aerospace and computer science were reviewed, as was common Industrial Advisory Board feedback with respect to computation. Examples in coeducating students in physical and discrete algorithm and model development were presented, followed by a discussion of how the aerospace curriculum might better promote computational thinking within existing courses as long as instructors buy into actually teaching such content alongside the mathematical methods traditionally presented. Options for accomplishing the computational thinking pedagogical goals in current as well as future curricula were reviewed, followed by a summary of related work in proposed and accomplished computing curricula. Each discussion indicates further change is needed to provide additional computer science content.

To conclude, it is important to reconsider and offer at least partial answers to two questions raised in the Introduction (Sec. I) and a third question related to the inevitable tradeoffs required to introduce new computing content in degree programs with hard credit limits. Note that these questions and answers are focused specifically on improving aerospace education in computational thinking and are relevant, whether subscribing to the rationalist, technocratic, or scientific paradigm of computer science.

1) What fundamental knowledge in computing do aerospace students need to acquire before graduation? Certainly, a baseline understanding of logic is a prerequisite to most computing concepts. With a primary goal of computational thinking and secondary goal of passing on knowledge in computer science and skills in software engineering that would be of use to as many aerospace students as possible, it is the author's opinion that repeat exposure to whatever material is covered is most critical. This may be achieved through integration of concepts across digital (discrete) and physical (continuous) models, and through capitalizing on opportunities to introduce improved awareness of data management and manipulation activities in the context of aerospace applications such as CAD, FEM, and control system simulation. Based on the author's past experiences, as new course credits become available to teach computational thinking, aerospace students will be most motivated to pursue computer science in the context of real embedded systems applications, aerospace faculty will be most excited about teaching the theory of computing and computational science, and the aerospace industry will seek students who are capable software developers/engineers. Ideally, elements cutting across all three tracks will ultimately be threaded into the curriculum, with additional content specific to each as possible, but this is necessarily a long-term goal.

2) What language(s) are appropriate to teach, especially in this age of automatic code generation, large code bases, big data, etc.? Although language was purposely not emphasized in this paper, it is important to recognize that different languages emphasize different aspects of computer science. MATLAB allows rapid examination of mathematical and numerical phenomena, whereas a low-level language such as C or C++ facilitates deployment of code on embedded hardware. Coverage of multiple languages may be advantageous, as this will help students realize computational thinking and object-oriented abstraction concepts are common across languages. Although code generation should be covered as an option, it is also important to help students recognize the value of computational thinking as a means to innovate beyond methods offered by packages that generate code.

3) Finally, perhaps the hardest question of all must be addressed: What content can be removed from current aerospace curricula to make space for new computing content? Whereas new models and methods are being developed in every pillar subdiscipline, fundamentals associated with the traditional aerospace pillars have not changed substantially. No one wants to give up any of their favorite course content. That said, since it is

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.1010193

the author's view that computing must be more comprehensively fit into required curricula, tradeoffs must be actively made, either by offering tracks that free credits (e.g., air versus space) or by compressing existing course content into fewer credits. Another option is to trust the students sufficiently to allow them to choose between a diverse set of technical elective options, as is done at MIT. If a student plans to pursue a career focused on a traditional pillar, substantial depth in that topic can be pursued as part of the aerospace degree program. Similarly, if a student is interested in pursuing a career that will require substantial knowledge of computing concepts, he/she can gain depth of knowledge in said computing concepts as part of the aerospace degree program, without double majoring.

As a closing thought, consider a time not so long ago, before MATLAB, when mathematical computations had to be done by hand or, at best, with the assistance of a low-level programming language such as Fortran or C. Students took the same math courses as are required today, calculus through differential equations, and the fundamentals of each subject were largely the same as they are today. How did each course expand over the last three decades to fill the time previously spent manually solving math problems? Then, how could instructors reclaim at least some of this valuable course time without loss of fundamental content to improve coverage of new computing concepts or to reinforce use of structured programming knowledge, e.g., by manually writing a differential equation solver rather than using MATLAB's built-in functionality? Only time will tell.

## References


[1] Etter, D. M., *Introduction to MATLAB for Engineers and Scientists*, Prentice Hall PTR, 1996.
[2] Chapman, S., *MATLAB Programming for Engineers*, Cengage Learning, 2007.
[3] Palm, W. J., *Introduction to MATLAB for Engineers (BEST Series)*, McGraw-Hill Higher Education, 1998.
[4] Arrington, E. A., "Structured Programming Education Requirements for Aerospace Engineering," *AIAA Aerospace Sciences Meeting*, AIAA Paper 2011-0466, 2011.
[5] Long, L. N., "The Critical Need for Software Engineering Education," *Crosstalk*, Vol. 21, No. 1, 2008, pp. 6–10.
[6] "Analytical Engine," Wikipedia, http://en.wikipedia.org/wiki/Analytical_Engine [retrieved 20 Dec. 2012].
[7] Geist, A., and Lucas, R., "Major Computer Science Challenges at Exascale," *International Journal of High Performance Computing Applications*, Vol. 23, No. 4, 2009, pp. 427–436.
[8] "Boolean Algebra," Wikipedia, http://en.wikipedia.org/wiki/Boolean_algebra_(logic) [retrieved 20 Dec. 2012].
[9] Boole, G., "An Investigation of the Laws of Thought," Prometheus Books, Amherst, NY, 2003, pp. 1–149.
[10] Akers, S. B., "Binary Decision Diagrams," *IEEE Transactions on Computers*, Vol. 100, No. 6, 1978, pp. 509–516.
[11] "Finite-State Machine," Wikipedia, http://en.wikipedia.org/wiki/Finite-state_machine [retrieved 20 Dec. 2012].
[12] "Turing Machine," Wikipedia, http://en.wikipedia.org/wiki/Turing_machine [retrieved 20 Dec. 2012].
[13] "Church–Turing Thesis," Wikipedia, http://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis [retrieved 20 Dec. 2012].
[14] "Computer Science," Wikipedia, http://en.wikipedia.org/wiki/Computer_science [retrieved 20 Dec. 2012].
[15] Eden, A. H., "Three Paradigms of Computer Science," *Minds and Machines, Special Issue on the Philosophy of Computer Science*, Vol. 17, No. 2, 2007, pp. 135–167.
[16] Wing, J., "Computational Thinking," *Communications of the ACM*, Vol. 49, No. 3, 2006, pp. 33–35. doi:10.1145/1118178
[17] Guzdial, M., "Education: Paving the Way for Computational Thinking," *Communications of the ACM*, Vol. 51, No. 8, 2008, pp. 25–27. doi:10.1145/1378704
[18] Quinlan, J. R., "Generating Production Rules from Decision Trees," *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Francisco, 1987, pp. 304–307.
[19] Laird, J. E., Newell, A., and Rosenbloom, P. S., "Soar: An Architecture for General Intelligence," *Artificial Intelligence*, Vol. 33, No. 1, 1987, pp. 1–64. doi:10.1016/0004-3702(87)90050-6
[20] Darmofal, D., "16.901 Computational Methods in Aerospace Engineering," Massachusetts Inst. of Technology, Cambridge, MA, Spring 2005.
[21] White, J., Peraire, J., Luca, D., Hadjiconstantinou, N., and Patera, A., "6.336J Introduction to Numerical Simulation," Massachusetts Inst. of Technology, Cambridge, MA, Fall 2003.
[22] Oblinger, D., and Oblinger, J., *Chapter 2: Is It Age or IT: First Steps Toward Understanding the Net Generation*, Educause, Boulder, CO, 2005, pp. 2.1–2.20.
[23] Hartman, J., Moskal, P., and Dziuban, C., *Chapter 6: Preparing the Academy of Today for the Learner of Tomorrow*, Educause, Boulder, CO, 2005, pp. 6.1–6.15.
[24] Barnes, K., Marateo, R., and Ferris, S., "Teaching and Learning with the Net Generation," 2007, http://uruguayeduca.edu.uy/Userfiles/P0001/File/Teaching_and_Learning_with_the_Net_Generation.pdf [retrieved 24 July 2014].
[25] Clinton, K., Purushotma, R., Robison, A., and Weigel, M., "Confronting the Challenges of Participatory Culture: Media Education for the 21st Century," *MacArthur Foundation Publication*, Vol. 1, No. 1, 2006, pp. 1–59.
[26] "Information Technology," Wikipedia, http://en.wikipedia.org/wiki/Information_technology [retrieved 24 Nov. 2012].
[27] "Computational Science," Wikipedia, http://en.wikipedia.org/wiki/Computational_science [retrieved 23 Dec. 2012].
[28] Meserole, J. S., and Moore, J. W., "What is System Wide Information Management (SWIM)?" *Aerospace and Electronic Systems*, Vol. 22, No. 5, 2007, pp. 13–19.
[29] "Science," Wikipedia, en.wikipedia.org/wiki/Science [retrieved 20 Dec. 2012].
[30] "Engineering," Wikipedia, en.wikipedia.org/wiki/Engineering [retrieved 20 Dec. 2012].
[31] "Implementation," Wikipedia, en.wikipedia.org/Implementation [retrieved 20 Dec. 2012].
[32] Rajkumar, R., Lee, I., Sha, L., and Stankovic, J., "Cyber-Physical Systems: The Next Computing Revolution," *Proceedings of 47th Design Automation Conference*, ACM/IEEE, Piscataway, NJ/New York, 2010, pp. 731–736.
[33] Atkins, E., and Bradley, J., "Aerospace Cyber-Physical Systems Education," *AIAA Infotech@Aerospace*, AIAA Paper 2013-4809, 2013.
[34] Lee, I., and Sokolsky, O., "Medical Cyber Physical Systems," *47th ACM/IEEE Design Automation Conference*, ACM/IEEE, Piscataway, NJ/New York, 2010, pp. 743–748.
[35] Jensen, J., Chang, D., and Lee, E., "A Model-Based Design Methodology for Cyber-Physical Systems," *7th International Wireless Communications and Mobile Computing Conference*, IEEE, Piscataway, NJ, 2011, pp. 1666–1671.
[36] Kim, K., "Collision Free Autonomous Ground Traffic: A Model Predictive Control Approach," *Proceedings of International Conference on Cyber-Physical Systems*, ACM/IEEE, Piscataway, NJ/New York, 2013, pp. 51–60.
[37] Wang, X., Hovakimyan, N., and Sha, L., "L1 Simplex: Fault-Tolerant Control of Cyber-Physical Systems," *Proceedings of the International Conference on Cyber-Physical Systems*, ACM/IEEE, Piscataway, NJ/New York, 2013, pp. 41–50.
[38] Han, K., Potluri, S., and Shin, K., "On Authentication in a Connected Vehicle: Secure Integration of Mobile Devices with Vehicular Networks," *Proceedings of the International Conference on Cyber-Physical Systems*, ACM/IEEE, Piscataway, NJ/New York, 2013, pp. 160–169.
[39] Krishna, M., and Shin, K., *Real-Time Systems*, Wiley, New York, 1999.
[40] "Real-Time Computing," Wikipedia, http://en.wikipedia.org/wiki/Real-time_computing [retrieved 5 Aug. 2013].
[41] "Proposed Changes to the Aerospace Engineering Degree Program in the College of Engineering Section in the Undergraduate Catalog 2014-2016," Univ. of Texas at Austin, Austin, TX, http://www.utexas.edu/faculty/council/2013-2014/uc_change/proposals_submitted/ENG_ASE.pdf [retrieved 28 Dec. 2013].
[42] "Binary Search Algorithm," Wikipedia, http://en.wikipedia.org/wiki/Binary_search_algorithm [retrieved 23 Dec. 2012].


Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.I010193

[43] "Bisection," Wikipedia, http://en.wikipedia.org/wiki/Bisection [retrieved 23 Dec. 2012].
[44] Alur, R., and Dill, D. L., "A Theory of Timed Automata," *Theoretical Computer Science*, Vol. 126, No. 2, 1994, pp. 183–235. doi:10.1016/0304-3975(94)90010-8
[45] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P. H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S., "The Algorithmic Analysis of Hybrid Systems," *Theoretical Computer Science*, Vol. 138, No. 1, 1995, pp. 3–34. doi:10.1016/0304-3975(94)00202-T
[46] Barr, V., and Stephenson, C., "Bringing Computational Thinking to K–12: What is Involved and What is the Role of the Computer Science Education Community?" *ACM Inroads*, Vol. 2, No. 1, 2011, pp. 48–54.
[47] Papastergiou, M., "Digital Game-based Learning in High School Computer Science Education: Impact on Educational Effectiveness and Student Motivation," *Computers and Education*, Vol. 52, No. 1, 2009, pp. 1–12. doi:10.1016/j.compedu.2008.06.004
[48] Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, E., Brennan, K., and Millner, A., "Scratch: Programming for All," *Communications of the ACM*, Vol. 52, No. 11, 2009, pp. 60–67. doi:10.1145/1592761
[49] Benson, B., Arfaee, A., Kim, C., Kastner, R., and Gupta, R., "Integrating Embedded Computing Systems into High School and Early Undergraduate Education," *IEEE Transactions on Education*, Vol. 54, No. 2, 2011, pp. 197–202. doi:10.1109/TE.2010.2078819
[50] Guzdial, M., "Education: Teaching Computing to Everyone," *Communications of the ACM*, Vol. 52, No. 5, 2009, pp. 31–33. doi:10.1145/1506409
[51] Vergaga, C., Urban-Lurain, M., Dresen, C., Coxen, T., MacFarlane, T., Frazier, K., and Briedis, D., "Aligning Computing Education with Engineering Workforce Computational Needs: New Curricular Directions to Improve Computational Thinking in Engineering Graduates," *39th IEEE Frontiers in Education*, IEEE, Piscataway, NJ, 2009, pp. 1–6. doi:10.1109/FIE.2009.5350463
[52] Long,, L., and Janrathitikam, O.,, "A New Software Engineering Course for Undergraduate and Graduate Students," *AIAA Infotech@Aerospace*, AIAA Paper 2010-3505, 2010.
[53] "Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering," IEEE Computer Society and Assoc. for Computing Machinery, Piscataway, NJ, 2004.
[54] Atkins, E., Green, J., Yi, J., Woo, H. B. J., Mok, A., and Xie, F., "The Tablesat Platform and its Verifiable Control Software," *AIAA Infotech@Aerospace*, AIAA Paper 2009-1844, 2009.
[55] Atkins, E., "A Project-Based Undergraduate Aerospace Sequence, with Embedded Computational Intelligence," *AIAA Infotech@Aerospace*, AIAA Paper 2011-1581, 2011.
[56] Crawley, E., Grietzer, E., Widnall, S., Hall, S., McManus, H., Hansman, J., Shea, J., and Landahl, M., "Reform of the Aeronautics and Astronautics Curriculum at MIT," *Journal of Engineering Education*, Vol. 83, No. 1, 1994, pp. 47–56.
[57] Lundqvist, K., and Srinivasan, J., "A First Course in Software Engineering for Aerospace Engineers," *IEEE Conference on Software Engineering Education and Training*, IEEE, Piscataway, NJ, 2006, pp. 77–86. doi:10.1109/CSEET.2006.5

M. Davies
*Associate Editor*

Downloaded by University of Michigan - Duderstadt Center on December 13, 2017 | http://arc.aiaa.org | DOI: 10.2514/1.1010193