A Multidisciplinary Approach Towards Computational Thinking for Science Majors*

Susanne Hambrusch
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907,
USA
seh@cs.purdue.edu

Christoph Hoffmann
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907,
USA
cmh@cs.purdue.edu

John T. Korb
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907,
USA
jtk@cs.purdue.edu

Mark Haugan
Department of Physics
Purdue University
West Lafayette, IN 47907,
USA
mph@physics.purdue.edu

Antony L. Hosking
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907,
USA
hosking@cs.purdue.edu

ABSTRACT

This paper describes the development and initial evaluation of a new course "Introduction to Computational Thinking" taken by science majors to fulfill a college computing requirement. The course was developed by computer science faculty in collaboration with science faculty and it focuses on the role of computing and computational principles in scientific inquiry. It uses Python and Python libraries to teach computational thinking via basic programming concepts, data management concepts, simulation, and visualization. Problems with a computational aspect are drawn from different scientific disciplines and are complemented with lectures from faculty in those areas. Our initial evaluation indicates that the problem-driven approach focused on scientific discovery and computational principles increases the student's interest in computing.

Categories and Subject Descriptors

K.3.2 [**Computer and Education**]: Computer and Information Science Education

General Terms

Design, Experimentation

Keywords

Computational thinking, curriculum, multi-disciplinary, computing for scientists.

*Work supported in part by the National Science Foundation under Grant No. CCF-0722210.

SIGCSE'09, March 3–7, 2009, Chattanooga, Tennessee, USA.

1. INTRODUCTION

Scientific research is now unthinkable without computing. The ubiquity of computerized instrumentation and detailed simulations generates scientific data in volumes that can no longer be understood without computation. For example, the Sloan Digital Sky Survey is mapping the heavens with a 2.5m dedicated telescope [15]. In the first 5 years of operation, it generated about 6TB of data. Similarly, large-scale simulations, of climate models, of fusion reactors, or of engineered materials, generate comparable data sets in weeks or days and could generate larger data sets but for the availability of faster computers. Such data volume has to be examined by computation [8]. In light of this evolution of science, future generations of scientists must engage computing and must understand what computer science can do for their work, much as they have to understand what mathematics already does for their work.

Here we describe a multi-disciplinary effort to develop a course on computational thinking for science majors. At Purdue, all science undergraduates must fulfill a computing requirement, generally by taking a CS course [9]. The new course has been developed by CS in collaboration with faculty in Physics, Biology, Chemistry, and Statistics [12]. It uses a problem-driven approach focusing on scientific discovery through computational methods grounded in computer science principles.

The development of introductory computing courses with a focus on non-majors, in particular science students, has recently received attention at a wide range of institutions. Related efforts include work at Carnegie-Mellon, Harvey Mudd, Princeton, and Winona State [3, 4, 13, 14, 17, 19]. Our effort is strongly influenced by the concept of computational thinking advocated by Wing et al. [5, 6, 18]. Two NSF-funded workshops on computational thinking related to the course development have been held at Purdue in 2007 and 2008, respectively [11]. The paper is organized as follows. Section 2 describes the course development philosophy and the material covered. Section 3 gives an overview of the course projects. Section 4 presents an initial evaluation and Section 5 concludes.

2. A COURSE FOR SCIENCE MAJORS

The course "Introduction to Computational Thinking" was developed by CS faculty with experience in teaching introductory courses in collaboration with science faculty. Course development

was guided by five main principles:

- *Lay the groundwork for computational thinking.*
  Computational thinking encompasses a diverse set of skills including formulating problems, making abstractions, and phrasing solutions in ways that can be satisfied computationally. These skills range from algorithms and data structures to presentation and visualization.

- *Present examples in a language familiar to the students.*
  Consider teaching the concept of a derivative in the calculus. A physics major will appreciate examples in mechanics, discussing speed and acceleration. A chemistry major might be intrigued by examples discussing reaction rates and dynamic equilibria. After those examples have been analyzed and understood, the teacher can then proceed to an abstract, mathematical view of the underlying structures. Similarly, it is our belief that science majors, conversant in the basics of the classical disciplines, will comprehend computational concepts more easily if those concepts can be motivated by examples from their scientific subdisciplines.

- *Teach in a problem-driven way.*
  This includes presenting only those programming language features that are used and are meaningful to students and demonstrating where and how computational principles are needed. For example, viewing a thermodynamic system from the computational perspective, as opposed to a purely descriptive view, naturally leads to randomized models and Monte Carlo techniques. This, in turn, motivates pseudo-random-number generation algorithms, and discussions of where else Monte Carlo methods have use and their limitations as a computational paradigm.

- *The programming language should right away allow a focus on computational principles.*
  The goal is that students are able to write meaningful programs in a short time and do not have to focus on – and struggle with – language details not meaningful to them at that point in time. In addition, extensive libraries used by the scientific community should be available. It is our view that Python satisfies these expectations.

- *Make effective use of visualization.*
  Visualization is an important element in computing and brings many quantitative scientific facts to life. Visualization should be used to better understand the scientific questions asked as well as to help understand computational principles and processes. While the benefits of multimedia learning have been well studied [7], visualizartion is often not used effectively in teaching computational concepts.

Three science disciplines – physics, chemistry, and bioinformatics – provided us with expectations on what they want students to learn:

- The Physics department uses the approach developed by Chabay and Sherwood in an introductory calculus-based physics course [1]. In the lab, students run and modify Python programs to model and visualize mechanical systems and fields in 3D using VPython [16]. Teaching programming and computational thinking is beyond the scope of this physics course. Physics faculty are interested in students gaining a more complete understanding of computation that can lead to (i) new computational opportunities for learning and research and (ii) a new perspective on the nature of and interplay between physics and applied mathematics by solving realistic problems computationally.

- The computational chemistry faculty are interested in students learning computational methods relevant in chemical research, in particular Monte Carlo, Simulated Annealing and Molecular Dynamics. In addition, being able to use and integrate existing Fortran programs and learning visualizing teachniques is viewed as important.

- For the area of bioinformatics and statistics, there is an interest in teaching the use of R for statistical computing and visualization [10], followed by learning how to program in a language for which bioinformatics software packages exist or can easily be integrated.

The following describes the material covered in the 15-week course, with two one-hour lectures and one two-hour lab per week:

I. Basic Programming Tools (6 weeks)

- Introduction to Python. Elementary values and data types.
- Straight line programs, assignments to variables, type conversion, math library.
- Strings, lists, and tuples. Vectors and arrays.
- Conditionals and loop structures.
- Plotting using MatPlotLib and 3D visualization in VPython.
- Functions, parameters, and scope. Recursion.

II. Computational Tools and Methods (6 weeks)

- Arithmetic and random numbers. Using NumPy. Examples of numerical stability and problem stability.
- Introduction to simulations and Monte Carlo methods.
- Computational Physics: Ideal gas and Ising Spin simulations; adapting a generic Demon algorithm and estimating parameters in a physical system.(1 week)
- Trees as a data structure, traversal and exploration.
- Introduction to graphs, graph operations using NetworkX, graphs in science applications.
- Bioinformatics: Modeling protein interactions using tree and graph representations. Visualizing graphs in Cytoscape and analyzing protein interactions using clustering techniques. (1 week)
- Grand challenges in scientific computing.

III. Looking Under the Hood at Computer Science (3 weeks)

- Object-oriented design. Use and design of classes, OO concepts. Dictionaries and spatial queries as examples.
- History of computer science.
- Limits of computing, intractability, computability.
- Future models of computation: DNA computing, quantum computing.

Python was chosen as the programming medium because of its interactive environment, ability to let novice programmers quickly write non-trivial programs, adoption by many scientific communities, and support for numerous specialist libraries. Python can be executed efficiently, making it a good vehicle not only for small-scale experimentation, but also for larger data sets and longer computational problems. The course includes teaching the basics of object-oriented design. We found this subject was quite natural to comprehend towards the end of the semester, after fluency in Python has been developed. On the other hand, recursion was considered challenging by the students. VPython and MatPlotLib were introduced early in the course. VPython allows creating sophisticated 3D visuals without having to learn the complexities of traditional libraries, such as OpenGL. The graphical programming done with MatPlotLib will serve students well in later courses and projects. Visualization clearly helped students to better understand the scientific questions asked in the projects and to better understand their programs. It confirms that visual computing is an engaging activity that is underutilized in many CS curricula.

The course was developed jointly with faculty in the other science departments who gave guest lectures. The guest lectures show how CS concepts arise when solving disciplinary problems. These lectures include concepts such as Maxwell's Demon and use state-of-the-art software, such as NetworkX for graph manipulation and CytoScape for visualizing protein interactions. The course material covered is to a large extent driven by the projects which are described in more detail in the next section.

3. PROJECT OVERVIEW

The course assigns four small-scale programming assignments and four projects. Almost all questions on the smaller programming assignments are preparation for the larger projects. Each project consists of a programming part and an experimental part (which for some projects use data culled from research). The programming part has an earlier deadline and the experimental part can be completed with the code the student writes in the programming part or with code made available. Interestingly, very few students decide to abandon their own code, even when it proves to be incorrect or its inefficiency does not allow completion of the experimental part of the project. The most common reason for inefficiency resulting in excessive running time is the use of $O(n^2)$ or $O(n^3)$ time algorithms when linear or sublinear solutions exist. Using functionalities provided by Python, such an unnecessary performance was at times not obvious to the novice programmer. All projects asked students to produce visualizations of computational results and provide a write-up on their observations.

1. *Manipulating Digital Audio.*
   Explore the generation and manipulation of digital sound. Students write and use several basic functions that represent sounds as a sequence of wave amplitudes. Students explore the creation of new operations on sounds. The project emphasizes arrays, loop structures, numeric data (including overflow and round off issues), and modularity through procedures. Experimentation encourages students to generate sounds with different kinds of waves (e.g., square and sawtooth) and to investigate music in different scales. Sounds are visualized using MatPlotLib functionality. First introduction to large data sets and computational complexity issues.

2. *Computational Experiments on Percolation in Grids.*
   Examine the spread of wild fire through a patchy field of dry grass, electricity through a surface of conductors and insulators, or how water soaks through a porous landscape. This project uses a two dimensional array to represent physical scenes and a single parameter to represent the probability that any node in the grid "percolates". By varying the parameter, generating random grids based on it, and simulating flow in those grids, students create plots to answer the question, "What is the smallest probability q at which a grid generated with probability q will percolate?" Flows through a grid are visualized in VPython. In addition to reinforcing loops, conditionals, and multi-dimensional arrays, this project uses random number generation and introduces recursive functions. Visualization is used in visualizing commutation as well as visualizing scientific results.

3. *Simulating Physical Systems.*
   This project elaborates on Monte Carlo methods as a way to understand the behavior of physical systems without resorting to a detailed dynamical simulation. It introduces the "demon algorithm" (from Maxwell's demon) and has students performing two experiments: (1) simulating an ideal gas to study the velocity distribution of particles of a gas in thermal equilibrium as a function of gas energy and temperature, and (2) using the 2-d Ising model to study the magnetization of a lattice of spins as a function of lattice energy and temperature and, so, explore an example of a phase transition. The project uses VPython and Matplotlib libraries to create 2-D and 3-D visualizations for experimental results. The project is natural for physics and chemistry majors. Class lectures giving the physics background are important.

4. *Analyzing Protein-Protein Interactions.*
   Analyze the results of large-scale experiments that characterize protein-protein interactions using graph models and predict the quality of the experimentally observed protein complexes. Students use the Python-based NetworkX library to manipulate and "clean" large graphs and are given code for finding clusters in graphs. Clustering results are related to scores generated from the publicly available Gene Ontology (GO) database. Large graphs are visualized and manipulated using Cytoscape, a high quality, open-source graph visualization tool used by bioinformatics researchers. The project needs class preparation on both graph manipulations as well as basic bioinformatics background.

Overall popular projects are percolation and the simulation of physical systems. Seeing different percolation algorithms detect different type of flows through a grid while graphs plot experimental probability results plays a role in making this a favorite project. The preference for the simulating physical systems project seems to be related to the large number of physics and chemistry majors who have seen this material in a physics course. The audio project is highly rated for the flexibility it gives to students and the creativity in creating new sounds. The project on protein interactions asks students to write specified algorithms operating on graphs using NetworkX functionality. The abstraction underlying the use of a graph representation and the exploration of graphs are challenging for students. It is the only project using real, large-scale data sets. Complete project descriptions can be found at http://secant.cs.purdue.edu/cs190c:projects.

Students have different views on the value of running and interpreting computational experiments. Some find the experimental part to be the most rewarding part of the projects, while others would prefer the project to to be done once the programs run. The computing requirement for science majors allows students to select among various courses and we believe students should have the option to choose a traditional programming course.

	Entry Survey			Exit Survey		
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.
Taking another CS course	1.62	1	0.92	2.46	3	1.45
Pursuing a career that requires programming skills	1.69	1	1.14	2.85	3	1.7

Table 1: Results for two entry and exit survey questions

4. EVALUATION

The objective of the course is providing a foundation of programming and computational principles that students can and will apply to scientific inquiry. All students enrolled to fulfill the college-wide computing requirement. Our goal is to get students interested in actively using computation in their major and to consider taking a second course to strengthen their computing skills. Our goal is not to turn science majors into CS majors.

Throughout the semester, the course explains what related material is covered in what CS course and we try to give students a sense of what they can learn in other CS courses. There was some discussion on what course sequence could lead to a minor in CS (which would also fulfill a multidisciplinary requirement for science majors). We encourage students to realize the benefit of writing simple programs using MatPlotlib or VPython in other projects and classes. More importantly, we want to provide students with the tools to write small Python programs for visualization and data analysis purposes. In follow-up studies we plan to track how many students end up taking another CS course.

The first offering of the course in spring 2008 had 13 students enrolled. 10 of the 13 students were physics majors, the other three were chemistry majors. About a third of the students double-majored in mathematics which they viewed as their secondary major. 10 of the 13 students were freshmen. Unfortunately, the class had only one female student (partially reflecting the enrollment in the corresponding majors). Looking at the background of the students, 27% had no programming experience, 40% had done some programming on their own, and 33% had taken a high school programming class.

Students taking the course completed an entry and an exit survey (anonymous id's allow us to link survey responses). Two relevant and interesting questions asked on both the entry and the exit survey are: "How would you rate your current interest in":

Q1: Taking another computer science course?

Q2: Pursuing a career that requires programming skills?

We offered a choice of five answers: 0) not interested, 1) somewhat uninterested, 2) undecided, 3) somewhat interested, and 4) very interested. Table 1 shows statistical results for the entry and the exit survey. Figure 1 shows a graphical comparison of each entry and exit response. Responses lying above the diagonal correspnd to an increased interest. The plot shows that about two thirds of the responses indicate an increase in interest, and no decrease was by more than one point.

Figure 2 shows the responses to both questions in the entry and exit survey given by each student (arrows from right to left show an increased interest). Not surprisingly there is a close relationship between the responses a student gave to the two questions.

We performed statistical tests even though the small number of students requires that these test results be interpreted with caution. A Wilcoxin signed rank test shows a significant difference in the points for both questions. A p-value of less than 0.05 indicates a significant difference. Question Q1 has a p-value of 0.032 and question Q2 has a p-value of 0.021. A Kruskal-Wallis nonparametric test shows that previous programming experience had no effect

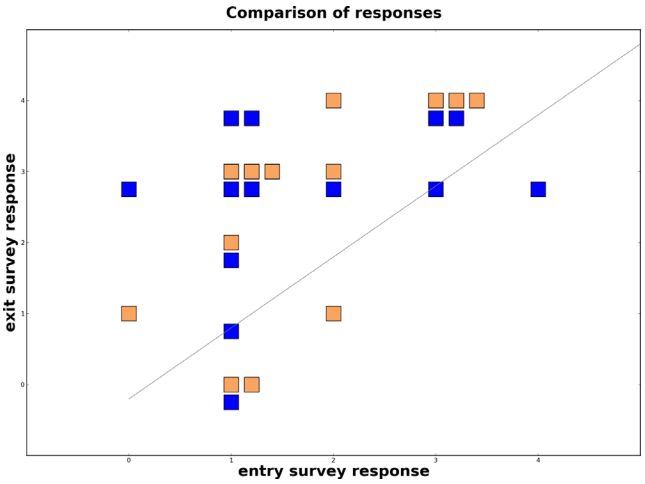

Figure 1: Comparison of entry and exit survey responses. Tan/light boxes show responses to Q1 "Taking another computer science course" and blue/dark boxes show responses to Q2 "Pursuing a career that requires programming skills"

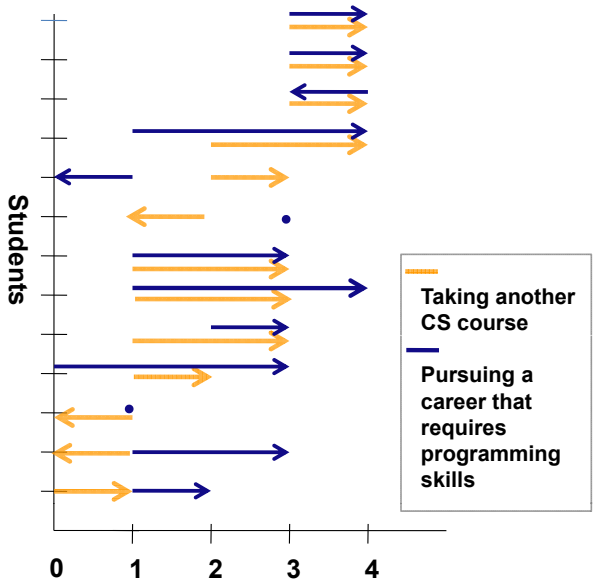

Figure 2: Responses to Q1 (tan/light) and Q2 (blue/dark) for each student.

on the interest in taking another CS course or pursuing a career requiring programming. The exit survey asked students whether they plan to take another CS course. Responses indicate that 60% of the students plan to take another CS course and 40% plan to minor in CS. Follow-up surveys will be conducted.

We point out that in our introductory programming courses for majors (Purdue students declare a major as freshmen), we see a decrease in interest in computer science and only 30% of CS freshmen complete a B.S. in computer science. The data reported by the Emerging Scholar Program carried out by a number of institutions also shows a decrease in interest, while showing that students in the program receive better grades and have a better experience [2]. The inability of introductory CS courses to maintain students' interest is viewed as one reason for the decreased enrollment in computer science.

From the feedback we have from students in the computational thinking class, it seems the problem-driven format of the course, the ability to quickly write programs meaningful to them, and the use of visualization tools all play a crucial role in the overall increased interest. Future instances of the course will assess this further and will track students with respect to additional computing courses taken.

5. CONCLUSIONS

We believe that interaction with science faculty is a critical element in designing an effective course in computational thinking for science majors. It is important for CS faculty to understand how the material taught can relate to material students will see in later science courses. Interaction with science faculty should be an on-going activity and not be limited to course development.

Based on our experience, Python is an excellent first language. It is used by many scientific disciplines, it allows us to teach modern concepts of programming, and it can be used interactively, giving students immediate feedback and giving them a convenient way to experiment with different constructs. Such concepts include recursion (which was considered challenging by the students) and object-oriented design (which they found natural). Moreover, visualization is an important element in computing and brings many quantitative scientific facts to life. VPython is a good vehicle because it focuses on a few simple basics in visualization and is learned quickly by doing.

A natural question is whether an introductory course should be taught in different versions to, say biology and physics majors. At this point we believe that the same first course can serve all science disciplines well. The majority of the scientific problems on our assignments are sufficiently elementary to be taught to all science majors. A few are more domain-specific. However, we find that it is valuable for students to work through the specifications of problems that lie outside of their specialist domain. Such situations prepare them for experiences they are likely to encounter later in life. While we do not plan to create different versions of our course, we consider the possibility to give students a choice on what projects to select.

We will build on this introductory course with a second course in which students collaborate in teams that include both science and CS majors. Working in a team is an important skill that should be practiced by all students. Collaborations between CS and the other sciences are increasingly common, so an early exposure to cultural differences across scientific disciplines is highly valuable.

6. ACKNOWLEDGMENTS

We thank Sagar Mittal, John Valko, and Rob Gevers for their valuable help on course and project development. We thank Olga Vitek and Sabre Kais for productive discussions in the areas of bioinformatics and computational chemistry.

7. REFERENCES

[1] R. W. Chabay and B. Sherwood. *Matter and Interactions, Vol. I: Modern Mechanics; Vol. II: Electric & Magnetic Interactions*. John Wiley and Sons, Hoboken, NJ, 2007.

[2] KD Evaluation Consultants. Evaluation of the emerging scholars program (ESP) in computer science. Technical report. 2008.

[3] T. J. Cortina. An introduction to computer science for non-majors using principles of computation. In I. Russell, S. M. Haller, J. D. Dougherty, and S. H. Rodger, editors, *Proceedings of the 38th ACM SIGCSE Technical Symposium on Computer Science Education*, 218–222. ACM, 2007.

[4] Z. Dodds, R. Libeskind-Hadas, C. Alvarado, and G. Kuenning. Evaluating a breadth-first CS 1 for scientists. In *SIGCSE '08: Proceedings of the 39th ACM SIGCSE Technical Symposium on Computer Science Education*, 266–270, 2008.

[5] M. Guzdial. Paving the way for computational thinking. *Commun. ACM*, 51(8):25–27, 2008.

[6] P. B. Henderson, T. J. Cortina, and J. M. Wing. Computational thinking. In *SIGCSE '07: Proceedings of the 38th ACM SIGCSE Technical Symposium on Computer Science Education*, 195–196, 2007.

[7] R. E. Mayer. *The Cambridge Handbook of Multimedia Learning*. Cambridge University Presss, New York, 2005.

[8] 2020 - Future of Computing. *Nature*, 440, March 2006.

[9] New science undergraduate curriculum. Purdue University, College of Science, 2007. http://www.science.purdue.edu/core/requirements2.asp.

[10] R: The R project for statistical computing, 2008. http://www.r-project.org/.

[11] NSF Workshops on Science Education in Computational Thinking ('07, '08). Purdue University. http://secant.cs.purdue.edu.

[12] Lectures and course material for "Introduction to computational thinking." Purdue University, Computer Science, 2008. http://secant.cs.purdue.edu/cs190c:start.

[13] R. Sedgewick and K. Wayne. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison Wesley, 2008.

[14] R. Sedgewick and K. Wayne. *Introduction to Computer Science*. Addison Wesley, in preparation.

[15] A. R. Thakar. The Sloan digital sky survey: Drinking from the fire hose. *Computing in Science & Engineering*, 10:9–12, January/February 2008.

[16] VPython: 3D programming for ordinary mortals, 2007. http://www.vpython.org/.

[17] G. Wilson, C. Alvarado, J. Campbell, R. Landau, and R. Sedgewick. CS-1 for scientists. *SIGCSE Bull.*, 40(1):36–37, 2008.

[18] J. M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, 2006.

[19] M. Zhang, E. Lundak, C.-C. Lin, T. Gegg-Harrison, and J. Francioni. Interdisciplinary application tracks in an undergraduate computer science curriculum. In *SIGCSE '07: Proceedings of the 38th ACM SIGCSE Technical Symposium on Computer Science Education*, 425–429, 2007.