University of Nebraska at Omaha
Course Syllabus

Department and Course Number	CSCI 1280
Course Title	Introduction to Computational Science
Course Coordinator	Victor Winter
Total Credits	3
Repeat for Credit?	No
Date of Last Revision	March 27, 2019

1. Course Description Information

1.1 Catalog description:

Introduction to Computational Science explores the role of computer science in scientific inquiry. Through the construction and analysis of block-based visual artifacts (e.g., pixel art and geometric patterns), this course aims to help students learn the essential thought processes used by computer scientists to solve problems, expressing those solutions as computer programs. When executed, these computer programs produce visual artifacts that can be displayed and interacted with using a variety of tools/software including LEGO Digital Designer, Minecraft, LDraw, 3D Builder, and Virtual Reality systems such as the HTC Vive.

1.2 Prerequisites of the course:

Math 1220 (or equivalent)

1.3 Overview of Content and purpose of the course:

Science, can be broadly defined as the intellectual and practical activity encompassing the systematic study of the structure and behavior of the physical and natural world through *observation and experiment*. The frontiers of modern science encompass phenomena for which computer-based modeling and simulation are playing an increasingly important role. In its 2005 report, the President's Information Technology Advisory Committee (PITAC) designated computational science as the "third pillar" of scientific inquiry. The PITAC report defines computational science as follows.

> *Computational science* is a rapidly growing multidisciplinary field that uses advanced computing capabilities to understand and solve complex problems. Computational science fuses three distinct elements:
>
> - Algorithms (numerical and non-numerical) and modeling and simulation software developed to solve science (e.g., biological, physical, and social), engineering, and humanities problems
> - Computer and information science that develops and optimizes the advanced system hardware, software, networking, and data management components needed to solve computationally demanding problems
> - The computing infrastructure that supports both the science and engineering problem solving and the developmental computer and information science

CSCI 1280 provides a gentle introduction to computational science in which computational thinking, principles of computing, and coding skills will be developed in the context of SML, a general-purpose functional programming language in which all computable functions can be expressed. A collection of libraries, collectively called Bricklayer, extend SML with a set of functions capable of creating a variety of geometric shapes including: lines, rings, circles, spheres, cones, cubes, and rectangular prisms. To facilitate the construction of artifacts composed of such shapes, the Bricklayer libraries provide computational

abstractions such as (1) iterators for property-based construction, and (2) a turtle for vector graphic-based construction. The Bricklayer libraries also provide a set of "show" functions enabling artifact display and interaction. Broadly speaking, Bricklayer artifacts can be classified as belonging to the domain of block-based (or cell-based) visual art. This visual domain is well-suited for exploring and developing an understanding of *patterns*. Patterns, play an important role in science. So much so, that the National Science Teachers Association (NSTA) have classified patterns as a crosscutting concept in the sciences.

In addition to the study of visual patterns, CSCI 1280 includes construction and experimentation with cell-based visual models and simulations suitable for exploring and analyzing fundamental laws and principles for a variety of natural and physical phenomena. The potential candidates that can be considered for such exploration is quite broad including (1) attractor/inhibitor systems such as *Turing patterns* which describe the way in which patterns in nature such as stripes and spots (and even appendages such as fingers and hands) can arise naturally out of a homogeneous, uniform state, as well as (2) cellular automata, a formalism originally invented by Stanislaw Ulam, to model and simulate John von Neumann's theoretical electromagnetic constructions whose ultimate goal was to create life[1]. Monte Carlo simulations are also explored as a means to obtain approximations of mathematical quantities such as percolation thresholds.

1.4 Unusual circumstances of the course:

None

2. Course Justification Information

2.1 Anticipated audience / demand:

This course is applicable for students in all majors who have no prior background in computer programming and who want to learn how to solve problems using computational techniques.

2.2 Indicate how often this course will be offered and the anticipated enrollment:

This course will be offered regularly in the Spring semester with an anticipated enrollment of around 30 students.

2.3 If it is a significant change to an existing course, please explain why it is needed:

This is a significant change to an existing course. It is intended to broaden participation in computing. It is also intended to supplement existing preparatory courses being used to get students ready for the more rigorous computer programming courses by focusing more on basic computational problem solving techniques rather than mastery of a certain language or technology.

3. List of performance objectives stated in learning outcomes in a student's perspective:

The student that successfully completes this course will:

3.1 Students will apply their knowledge of computational thinking in order to develop algorithms of simple complexity related to the construction of geometric patterns.

3.2 Students will apply their knowledge of computational thinking in order to develop algorithms of moderate complexity related to the construction of geometric patterns.

3.3 Students will be able to design computational solutions to problems whose solutions require the creation of geometric patterns of simple complexity.

3.4 Students will be able to design computational solutions to problems whose solutions require the creation of geometric patterns of moderate complexity.

[1] John von Neumann defined life as a creation (as a being or organism) which can reproduce itself and simulate a Turing machine.

3.5 Students will apply their knowledge of problem decomposition techniques in order to manage problem complexity.

3.6 Students will be able to employ programming as a creative tool.

3.7 Students will have increased spatial abilities – that is, the ability to understand, reason, and remember the spatial relations among 2D and 3D artifacts.

3.8 Students will have an increased ability to recognize the attributes (e.g., symmetry and repetition) possessed by visual patterns.

3.9 Students will have a semi-formal understanding of the semantics of the core of the SML programming language.

4. Content and Organization

List the major topics central to this course:

Digital Literacy

1. Files and Folders (saving, backing up, transferring, renaming, extensions)
2. Text editor basics (e.g., copy, cut, paste, column edit, search, replace, undo, redo, select)
3. Representing Discrete Values
   (a) Base n for n = 2, 10, 16
4. RGB color representations
   (a) Hex representations
   (b) Base 10 representations
5. Floating point numbers

Mathematics

1. Coordinate Systems
   (a) Cartesian
2. Functions
   (a) Transformations
   (b) Mappings
   (c) Projections
3. Probability
4. Operators
   (a) Logical
   (b) Relational
   (c) Arithmetic
5. Sets
6. Graphs

7. Sequences
    (a) Arithmetic Progressions
    (b) Geometric Progressions
8. Summations
9. Induction
10. Recurrence relations and closed forms

Computer Science

1. Syntax
    (a) The basics of regex
    (b) The basics of context-free grammars
2. Language Constructs
    (a) Values
        i. Integer
        ii. Real
        iii. String
        iv. Function
        v. Aggregations
            A. Tuple
            B. List
    (b) Expressions
        i. Arithmetic
        ii. Boolean
    (c) Declarations
        i. Variables
        ii. Functions
            A. Declarations
            B. Formal and actual parameters
            C. Curried functions
3. Control flow
    (a) Conditional Expressions
    (b) Sequential Composition
    (c) Iterators
4. Scope
5. Types
6. Design Concepts and Techniques
    (a) Verification and validation
    (b) Overwriting
    (c) Abstraction

(d) Generalization
(e) Composition/Decomposition
(f) Debugging techniques
(g) Code structuring
(h) Reliable software development processes
(i) Correctness-preserving transformations
    i. Refactoring
    ii. Optimization

Patterns

1. Static patterns
2. Growth patterns (sequences)
    (a) Arithmetic Patterns
    (b) Geometric Patterns
    (c) Evolutionary Patterns
3. Symmetry

Modeling and Simulation

1. Random Numbers
2. Cellular Automata and Elementary Cellular Automata
3. Attractor/Inhibitor Systems
4. Using Computers to Obtain Mathematical Approximations

5. Teaching Methodology Information

5.1 Methods

Teaching methods include (1) short lectures and demonstrations, (2) videos and on-line reading, (3) interactive web apps, (4) discussions, (5) hands-on in-class coding activities, (6) group and individual pixel art projects, and (7) group and individual art shows.

5.2 Student role in the course

The student is expected to attend class, complete in-class coding activities, participate in discussions, give Art Show presentations, complete assigned homework problems, and pass a midterm and final examination.

5.3 Contact hours:

3 hours per week

6. Evaluation Information

6.1 Describe the typical types of student projects that will be the basis for evaluating student performance:

Students will be evaluated on the code they develop, their presentations, as well as their performance on exams. Given the artistic possibilities of LEGO, a successful type of project-based student-directed assignment is an open-ended "Art Show". In an Art Show, students can work in groups or individually to create any LEGO structure they desire and the resulting artifact and the code that created it will be displayed for the entire class to see. Such art shows have proven to be extremely motivating.

6.2 Basis for determining the final grade:

Homework and Presentations	50%
Midterm	25%
Final	25%

6.3 Grading scale

Grades will range from A to F, with the specifics given by the table shown below. This information will be contained in the course outline and made available on the first day of class.

Grade	**Point Value**
A+	$96 \leq x \leq 100$
A	$92 \leq x < 96$
A-	$89 \leq x < 92$
B+	$86 \leq x < 89$
B	$82 \leq x < 86$
B-	$79 \leq x < 82$
C+	$76 \leq x < 79$
C	$72 \leq x < 76$
C-	$69 \leq x < 72$
D+	$66 \leq x < 69$
D	$62 \leq x < 66$
D-	$59 \leq x < 62$
F	$x < 59$

7. Resources

1. Textbooks(s) or other required readings used in the course:

   The textbooks given below cover some of the foundational material of the course and will be provided to the students as in PDF form free of charge.

   [1] K. H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill, New York, NY, USA, 6^{th} edition, 2007.

   [2] J. D. Ullman. *Elements of ML Programming.* Prentice Hall, 1998.

2. Other student suggested reading materials.

   [1] J. S. Conery. *Explorations in Computing - An Introduction to Computer Science.* Taylor and Francis Group, 2011.

   [2] T. Gaddis. *Starting Out with Programming Logic and Design.* Pearson, 2013.

   [3] J. Kun. *A Programmer's Introduction to Mathematics.* Pimbook.org, 2018.

   [4] J. D. Stone. *Algorithms for Functional Programming.* Springer-Verlag, 2018.

3. Current bibliography and other resources:

   [1] J. A. Adam. *Mathematics in Nature: Modeling Patterns in the Natural World.* Princeton University Press, 2006.

   [2] P. Ball. *The Self-Made Tapestry - Pattern Formation in Nature.* Oxford University Press, 1999.

   [3] P. Ball. *Patterns in Nature: Why the Natural World Looks the Way It Does.* The University of Chicago Press, 2016.

   [4] R. Bird. *Thinking Functionally with Haskell.* Cambridge University Press, 2015.

   [5] A. Deutsch and S. Dormann. *Cellular Automaton Modeling of Biological Pattern Formation.* Birkhauser, 2^{nd} edition, 2017.

   [6] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

   [7] M. R. Hansen and H. Rischel. *Introduction to Programming using SML.* Addison-Wesley, 1999.

   [8] A. S. Khot and R. K. Mishra. *Learning Functional Data Structures and Algorithms.* Packt Publishing, 2017.

   [9] J. P. Mueller. *Functional Programming for Dummies.* John Wiley & Sons, 2019.

   [10] M. du Sautoy. *Symmetry - A Journey into the Patterns of Nature.* HarperCollins, 2008.

   [11] D. K. Washburn and D. W. Crowe. *Symmetries of Culture: Theory and Practice of Plane Pattern Analysis.* University of Washington Press, 1998.

   [12] S. Wolfram. *A New Kind of Science.* Wolfram Media, Inc., 2002.

8. Estimate Computer Science Accreditation Board (CSAB) Category Content (class time in hours):

CSAB Category	Core	Advanced
Data structures	4	0
Computer organization and architecture	0	0
Algorithms and software design	16	0
Concepts of programming languages	8	0

9. Oral and Written Communications:

Every student is required to submit at least 0 written reports (not including exams, tests, quizzes, or commented programs) to typically 0 pages and to make 4 oral presentations of typically 4 minutes duration.

Approximately 4 Art Show assignments will be given during the semester. For each Art Show, students will be expected to give a short presentation (4 minutes) to the class in which they describe the artifact they have submitted to the Art Show. For example, what inspired their creation? What is interesting about their creation? What was challenging about their creation? When appropriate the instructor will direct questions to the class relating to how an artifact was created (e.g., what kind of algorithm or construction process was employed?). The instructor will also provide a light critique of the student's code.

10. Social and Ethical Issues:

No coverage

11. Theoretical content:

The primary focus of CSCI 1280 is on understanding patterns and expressing patterns in code. At the heart of this problem lies the management of complexity. The understanding of a pattern must be intellectually manageable. Similarly, the code which creates a pattern must also be intellectually manageable.

Pattern Analysis techniques:	15 hours
Code restructuring techniques:	15 hours

12. Problem analysis:

Students are exposed to problem analysis and problem solving through coding examples and assignments. In CSCI 1280 problem analysis and problem solving center on challenges relating to the construction of 2D/3D artifacts. There are three major areas in which the problem analysis and problem solving skills of students will be engaged: (1) construction processes, (2) group coordination, planning, and communication, and (3) artifact composition/decomposition techniques.

13. Solution design:

A sequence of programming assignments provides students with hands-on experience in the design and implementation of functional programs.

14. Change History

Date	**Change**	**By whom**	**Comments**
03/29/2019	Submission of proposal	Winter	Submitted to CSCI

A Gen Ed Science

SLO 1	*demonstrate a broad understanding of the fundamental laws and principles of science and interrelationships among science and technology disciplines*

Science, can be broadly defined as the intellectual and practical activity encompassing the systematic study of the structure and behaviour of the physical and natural world through *observation and experiment*.

The frontiers of modern science encompass phenomena for which computer-based modeling and simulation are playing an increasingly important role. As a result, *computational science* is now considered to be the "third pillar" of scientific inquiry [1]. Aligning with this perspective, CSCI 1280 includes construction and experimentation with cell-based visual models and simulations suitable for exploring and analyzing fundamental laws and principles for a variety of natural and physical phenomena. The potential candidates that can be considered for such exploration is quite broad including (1) attractor/inhibitor systems such as *Turing patterns* which describe the way in which patterns in nature such as stripes and spots (and even appendages such as fingers and hands) can arise naturally out of a homogeneous, uniform state, as well as (2) cellular automata, a formalism originally invented by Stanislaw Ulam, to model and simulate John von Neumann's theoretical electromagnetic constructions whose ultimate goal was to create life[2].

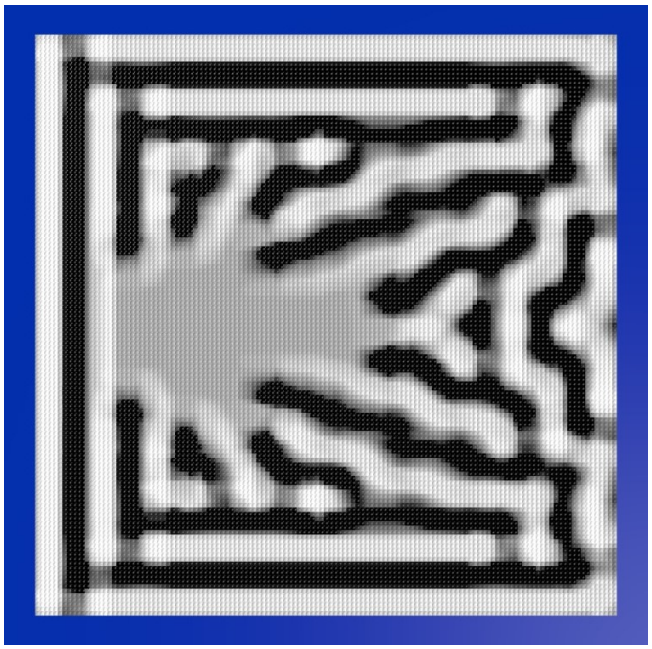

Figure 1: A Turing-like pattern created in Bricklayer. Note the pattern has horizontal reflection symmetry and appears hand-like.

Successful completion of this course provides the following.

1. The student will be able to formulate hypothesis, design experiments to test hypothesis, and collect and interpret data obtained from:
   (a) attractor/inhibitor models and simulations, and
   (b) cellular automata-based models and simulations.

[2]John von Neumann defined life as a creation (as a being or organism) which can reproduce itself and simulate a Turing machine.

2. The student will have:

   (a) an understanding of the increasingly important role played by computer-based modeling and simulation in scientific inquiry and the discovery of scientific laws.

SLO 2	*demonstrate a broad understanding of various natural and/or physical phenomena that surround and influence our lives*

Computational science views a *system* (S) as a potential source of data, an *experiment* (E) as a process of extracting data from a system by exerting it through its inputs, and a *model* (M) of S as anything to which E can be applied in order to answer questions about S.

While some consider computer science to be an artificial science because it studies phenomena surrounding computers and related digital technologies, in its essence, computer science studies information processes both artificial and natural [2]. Information processes take data inputs and produce outputs just as physical and biological processes take stimuli and produce responses. There has been an increasing realization that many natural phenomena are, fundamentally, information processes. DNA encodes information about living organisms, and quantum electrodynamics had been described as nature's computational method for combining quantum particle interactions [3]. To study information processes, there is a need to study the foundational computing principles that underlie these processes. The study of these *computing principles* is a central focus of CSCI 1280.

Scientists have come to increasingly rely on computational models to study and describe complex phenomena in a variety of fields including: climate science, particle physics and system biology. Many problems in these fields are not amenable to theoretical solutions and can only be addressed through large scale computations and simulations. *Percolation theory*[8] provides an example of the importance of computational models and simulations. A representative question in Percolation theory is as follows. Assume that a liquid is poured on top of some porous material. Will the liquid be able to make its way from hole to hole and reach the bottom? Percolation is an important scientific model because of its numerous applications to chemistry, biology, statistical physics, epidemiology, and materials science. For example, a composite system comprised of metallic (open) and insulating (blocked) materials is an electrical conductor if there is a metallic path from top to bottom, with full sites conducting. The creation of 2D and 3D percolation models can be useful in studying this phenomenon. Monte Carlo simulations can be used to approximate *percolation thresholds* (i.e., the probability that a system with a given density percolates). This is an example of a calculation for which no mathematical solution has yet been derived and for which computational models can provide important information. Two-dimensional models and simulations involving experimentation with and analysis of percolation thresholds lie well within the technical reach and scope of CSCI 1280.

Successful completion of this course provides the following.

1. The student will be able to:

   (a) conduct experiments involving Monte Carlo simulations of natural phenomena and draw conclusions from such simulations, and

   (b) use Monte Carlo simulations in support of mathematical analysis and to obtain mathematical results.

2. The student will have:

   (a) an increased understanding of information processes and phenomena surrounding computers,

   (b) an increased understanding of the computational principles that underlie information processes, and

   (c) an increased understanding of how computer-based models and simulations can be used to study various natural phenomena.

SLO 3	*describe how scientists approach and solve problems including an understanding of the basic components and limitations of the scientific method*

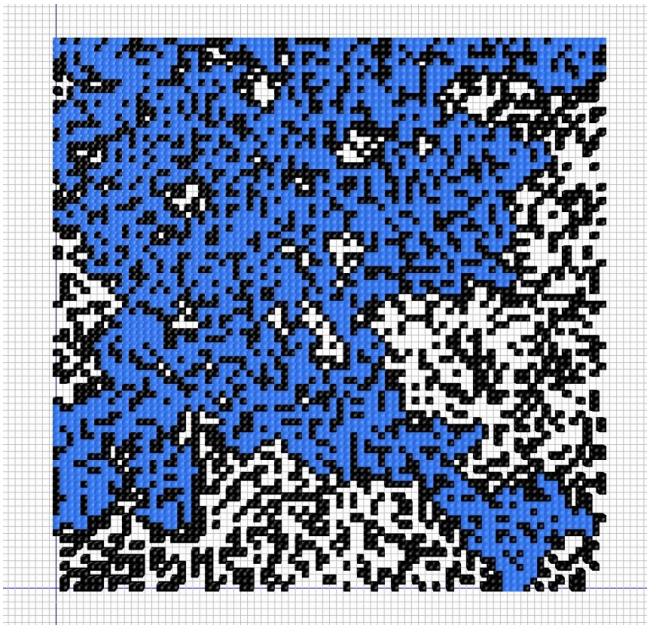

Figure 2: A randomly generated Bricklayer 2D cell structure that percolates.

A foundational component of CSCI 1280 is the introduction of computational thinking [9]. Computational thinking is broadly defined as the process by which computer scientists approach and solve problems. Computational thinking is a way of solving problems by mapping a natural problem into an algorithmic model that is amenable to a computational solution. This is a form of empirical inquiry [5] similar to how scientists would approach a problem by devising hypotheses and designing experiments to validate the hypothesis. The implementation of the program on a computational device becomes the experiment that validates the algorithmic model. It is worth noting that computational errors arising from faulty programming can produce incorrect results and have been known to lead to retractions [4]. Formal training of future scientists in the computer science disciplines can reduce the risk of such errors. CSCI 1280 provides a starting point for studying the principles of sound software development practices.

Successful completion of this course provides the following.

1. The student will be able to:
   (a) map a natural problem to an algorithmic model, and
   (b) employ hypothesis and experimentation-based techniques to validate a variety of algorithmic models and their implementations in software.
2. The student will have:
   (a) an increased understanding of computational thinking and its role in problem solving,
   (b) an increased understanding of sound software development practices, and
   (c) an increased understanding of the limits of computation.

SLO 4	*solve problems and draw conclusions based on scientific information and models, using critical thinking and qualitative and quantitative analysis of data and concepts in particular to distinguish reality from speculation*

The second law of thermodynamics states that the total entropy of an isolated system can never decrease over time. Or stated in layman's terms, "the universe tends towards disorder". Thus, we expect meaning in the patterns we see because, in a random universe, it takes energy to create order. So when we see a particular pattern, we expect that through investigation we can identify the force that caused it. This line of reasoning suggests that patterns play an important (possibly central) role in science. "Futurist and entrepreneur Ray Kurzweil considers pattern recognition so important that in his recent book, *How to Create A Mind*, he argued that pattern recognition and intelligence are essentially the same thing. Expertise, in essence, is the familiarity of patterns of a specific field."[7]

The National Science Teachers Association (NSTA) also recognizes the importance of patterns and has classified patterns as a concept crosscutting all the sciences.

- "Noticing patterns is often a first step to organizing phenomena and asking scientific questions about why and how the patterns occur."[6]
- "Once patterns and variations have been noted, they lead to questions; scientists seek explanations for observed patterns and for the similarity and diversity within them."[6]

Such statements provide support for an argument that patterns provide a catalyst for *observation and experiment*, the essence of science.

A pattern can be defined as a perceived regularity in a particular domain (e.g., visual domain). A considerable body of work is devoted to the classification of visual patterns according to their mathematical properties. Such classification begins by determining whether the pattern (i.e., the perceived regularity) occurs in one, two, or three dimensions. Further classification of a pattern is based on the symmetries it possesses (e.g., rotation, reflection, translation). Bricklayer provides an environment well-suited for exploring and developing an understanding of visual *patterns*. Such patterns can have natural or mathematical origins. *Mystique*, a web app which is part of the Bricklayer ecosystem, has been specifically designed to develop and strengthen the understanding of symmetry. Furthermore, the effort involved in writing a Bricklayer program that creates a pattern, reduces as the programmers understanding of the pattern increases.

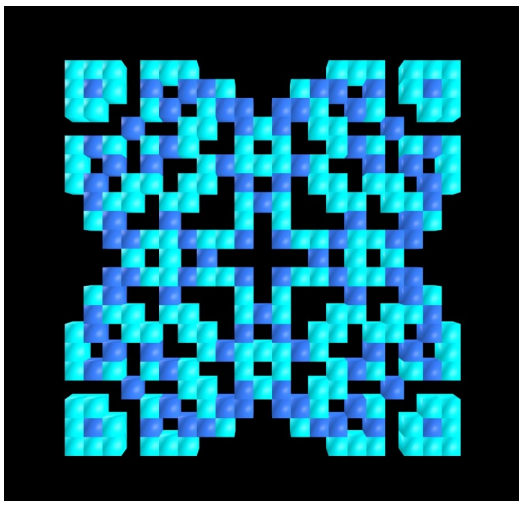

Figure 3: A randomly generated Bricklayer 2D pattern possessing horizontal and vertical reflection symmetry as well as 2-fold and 4-fold rotational symmetry.

Successful completion of this course provides the following.

1. The student will be able to:
   (a) recognize reflectional and rotational symmetry in visual artifacts,

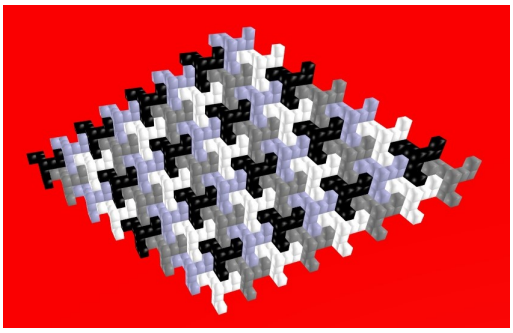

Figure 4: A 3D tessellation created in Bricklayer.

(b) classify a variety of visual patterns according to their mathematical properties, and

(c) use symmetry-based information and analysis to solve problems relating to the construction of a variety of visual patterns in code.

2. The student will have:

(a) an increased understanding of relationships between visual patterns and their mathematical characteristics, and

(b) an increased understanding of the role of pattern recognition as a driver of scientific inquiry based on computational simulations and analysis.

B What are the proposed method(s) for assessment of this course for the Natural Science SLOs?

CSCI 1280 includes an in-depth study of patterns and symmetry. A number of assignments as well as quizzes will be given on these topics.

In CSCI 1280, students will also be given assignments where involving experimentation with modeling and simulation. In addition to computational experimentation, student will be asked to write brief summaries of their experiments and findings.

References

[1] P. I. T. A. Committee. *Computational Science: Ensuring America's Competitiveness*. PITAC, 2005.

[2] P. J. Denning. Is computer science science? *Commun. ACM*, 48(4):27–31, Apr. 2005.

[3] P. J. Denning. Computing is a natural science. *Commun. ACM*, 50(7):13–18, July 2007.

[4] Z. Merali. Computational science: ...Error. *Nature*, 467(7317):775–777, 2010.

[5] A. Newell and H. A. Simon. Computer science as empirical inquiry: Symbols and search. *Commun. ACM*, 19(3):113–126, Mar. 1976.

[6] N. R. C. of the National Academes. *A Framework for K-12 Science Eduction - Practices, Crosscutting Concepts, and Core Ideas*. The National Academies Press, 2012.

[7] G. Satell. The Science of Patterns. *Forbs*, May 2015.

[8] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*. CRC Press, 1994.

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

C Relationship to ACM CS Curriculum Matrix

The ACM Curriculum Matrix uses three levels of mastery, defined as:

- *Familiarity*: The student understands what a concept is or what it means. This level of mastery concerns a basic awareness of a concept as opposed to expecting real facility with its application. It provides an answer to the question "What do you know about this?"
- *Usage*: The student is able to use or apply a concept in a concrete way. Using a concept may include, for example, appropriately using a specific concept in a program, using a particular proof technique, or performing a particular analysis. It provides an answer to the question "What do you know how to do?"
- *Assessment*: The student is able to consider a concept from multiple viewpoints and/or justify the selection of a particular approach to solve a problem. This level of mastery implies more than using a concept; it involves the ability to select an appropriate approach from understood alternatives. It provides an answer to the question "'Why would you do that?"

C.1 Knowledge Area: CN - Computational Science

Abstraction is a fundamental concept in computer science. A principal approach to computing is to abstract the real world, create a model that can be simulated on a machine. The roots of computer science can be traced to this approach, modeling things such as trajectories of artillery shells and the modeling cryptographic protocols, both of which pushed the development of early computing systems in the early and mid-1940's.

Modeling and simulation of real world systems represent essential knowledge for computer scientists and provide a foundation for computational sciences. Any introduction to modeling and simulation would either include or presume an introduction to computing. In addition, a general set of modeling and simulation techniques, data visualization methods, and software testing and evaluation mechanisms are also important.

CN/Introduction to Modeling and Simulation

Explain the concept of modeling and the use of abstraction that allows the use of a machine to solve a problem.	usage
Describe the relationship between modeling and simulation, i.e., thinking of simulation as dynamic modeling.	usage
Create a simple, formal mathematical model of a real-world situation and use that model in a simulation.	familiarity
Differentiate among the different types of simulations, including physical simulations, human-guided simulations, and virtual reality.	familiarity
Describe several approaches to validating models.	familiarity
Create a simple display of the results of a simulation.	usage

CN/Modeling and Simulation

Explain and give examples of the benefits of simulation and modeling in a range of important application areas.	familiarity
Demonstrate the ability to apply the techniques of modeling and simulation to a range of problem areas.	familiarity
Explain the constructs and concepts of a particular modeling approach	familiarity
Explain the difference between validation and verification of a model; demonstrate the difference with specific examples[a] [a]Verification means that the computations of the model are correct. If we claim to compute total time, for example, the computation actually does that. Validation asks whether the model matches the real situation.	familiarity
Verify and validate the results of a simulation.	familiarity
Evaluate a simulation, highlighting the benefits and the drawbacks.	familiarity
Choose an appropriate modeling approach for a given problem or situation.	familiarity
Compare results from different simulations of the same situation and explain any differences.	not covered
Infer the behavior of a system from the results of a simulation of the system.	familiarity
Extend or adapt an existing model to a new situation.	not covered

CN/Processing

Explain the characteristics and defining properties of algorithms and how they relate to machine processing.	familiarity
Analyze simple problem statements to identify relevant information and select appropriate processing to solve the problem.	assessment
Identify or sketch a workflow for an existing computational process such as the creation of a graph based on experimental data.	not covered
Describe the process of converting an algorithm to machine-executable code.	assessment
Summarize the phases of software development and compare several common lifecycle models.	not covered
Explain how data is represented in a machine. Compare representations of integers to floating point numbers. Describe underflow, overflow, round off, and truncation errors in data representations.	familiarity
Apply standard numerical algorithms to solve ODEs and PDEs. Use computing systems to solve systems of equations.	not covered
Describe the basic properties of bandwidth, latency, scalability and granularity.	not covered
Describe the levels of parallelism including task, data, and event parallelism.	not covered
Compare and contrast parallel programming paradigms recognizing the strengths and weaknesses of each.	not covered
Identify the issues impacting correctness and efficiency of a computation.	not covered
Design, code, test and debug programs for a parallel computation.	not covered

CN/Interactive Visualization

Compare common computer interface mechanisms with respect to ease-of-use, learnability, and cost.	not covered
Use standard APIs and tools to create visual displays of data, including graphs, charts, tables, and histograms.	not covered
Describe several approaches to using a computer as a means for interacting with and processing data.	not covered
Extract useful information from a dataset.	not covered
Analyze and select visualization techniques for specific problems.	familiarity
Describe issues related to scaling data analysis from small to large data sets.	familiarity

CN/Data, Information, and Knowledge

Identify all of the data, information, and knowledge elements and related organizations, for a computational science application.	not covered
Describe how to represent data and information for processing.	familiarity
Describe typical user requirements regarding that data, information, and knowledge.	not covered
Select a suitable system or software implementation to manage data, information, and knowledge.	not covered
List and describe the reports, transactions, and other processing needed for a computational science application.	not covered
Compare and contrast database management, information retrieval, and digital library systems with regard to handling typical computational science applications.	not covered
Design a digital library for some computational science users/societies, with appropriate content and services.	not covered

Numerical Analysis

Define error, stability, machine precision concepts and the inexactness of computational approximations.	familiarity
Implement Taylor series, interpolation, extrapolation, and regression algorithms for approximating functions.	not covered
Implement algorithms for differentiation and integration.	not covered
Implement algorithms for solving differential equations.	not covered

C.2 Knowledge Area: DS - Discrete Structures

Discrete structures are foundational material for computer science. By foundational we mean that relatively few computer scientists will be working primarily on discrete structures, but that many other areas of computer science require the ability to work with concepts from discrete structures. Discrete structures include important material from such areas as set theory, logic, graph theory, and probability theory.

The material in discrete structures is pervasive in the areas of data structures and algorithms but appears elsewhere in computer science as well. For example, an ability to create and understand a proof–either a formal symbolic proof or a less formal but still mathematically rigorous argument–is important in virtually every area of computer science, including (to name just a few) formal specification, verification, databases, and cryptography. Graph theory concepts are used in networks, operating systems, and compilers. Set theory concepts are used in software engineering and in databases. Probability theory is used in intelligent systems, networking, and a number of computing applications.

Given that discrete structures serves as a foundation for many other areas in computing, it is worth noting that the boundary between discrete structures and other areas, particularly Algorithms and Complexity, Software Development Fundamentals, Programming Languages, and Intelligent Systems, may not always be crisp. Indeed, different institutions may choose to organize the courses in which they cover this material in very different ways. Some institutions may cover these topics in one or two focused courses with titles like "discrete structures" or "discrete mathematics," whereas others may integrate these topics in courses on programming, algorithms, and/or artificial intelligence. Combinations of these approaches are also prevalent (e.g., covering many of these topics in a single focused introductory course and covering the remaining topics in more advanced topical courses).

DS/Sets, Relations, and Functions

Explain with examples the basic terminology of functions, relations, and sets.	familiarity
Perform the operations associated with sets, functions, and relations.	familiarity
Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context.	familiarity

DS/Basic Logic

Convert logical statements from informal language to propositional and predicate logic expressions.	usage
Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae and computing normal forms.	not covered
Use the rules of inference to construct proofs in propositional and predicate logic.	not covered
Describe how symbolic logic can be used to model real-life situations or applications, including those arising in computing contexts such as software analysis (e.g., program correctness), database queries, and algorithms.	not covered
Apply formal logic proofs and/or informal, but rigorous, logical reasoning to real problems, such as predicting the behavior of software or solving problems such as puzzles.	usage
Describe the strengths and limitations of propositional and predicate logic.	not covered

DS/Basics of Counting

Apply counting arguments, including sum and product rules, inclusion-exclusion principle and arithmetic/geometric progressions.	usage
Apply the pigeonhole principle in the context of a formal proof.	not covered
Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application.	familiarity
Map real-world applications to appropriate counting formalisms, such as determining the number of ways to arrange people around a table, subject to constraints on the seating arrangement, or the number of ways to determine certain hands in cards (e.g., a full house).	not covered
Solve a variety of basic recurrence relations.	usage
Analyze a problem to determine underlying recurrence relations.	usage
Perform computations involving modular arithmetic.	usage

DS/Graphs and Trees

Illustrate by example the basic terminology of graph theory, as well as some of the properties and special cases of each type of graph/tree.	familiarity
Demonstrate different traversal methods for trees and graphs, including pre-, post-, and in-order traversal of trees.	not covered
Model a variety of real-world problems in computer science using appropriate forms of graphs and trees, such as representing a network topology or the organization of a hierarchical file system.	familiarity
Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting.	not covered
Explain how to construct a spanning tree of a graph.	not covered
Determine if two graphs are isomorphic.	not covered

DS/Discrete Probability

Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance.	familiarity
Differentiate between dependent and independent events.	familiarity
Identify a case of the binomial distribution and compute a probability using that distribution.	not covered
Apply Bayes theorem to determine conditional probabilities in a problem.	not covered
Apply the tools of probability to solve problems such as the average case analysis of algorithms or analyzing hashing.	not covered
Compute the variance for a given probability distribution.	not covered
Explain how events that are independent can be conditionally dependent (and vice-versa). Identify real-world examples of such cases.	not covered

C.3 Knowledge Area: PL - Programming Languages

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. In the course of a career, a computer scientist will work with many different languages, separately or together. Software developers must understand the programming models underlying different languages and make informed design choices in languages supporting multiple complementary approaches. Computer scientists will often need to learn new languages and programming constructs, and must understand the principles underlying how programming language features are defined, composed, and implemented. The effective use of programming languages, and appreciation of their limitations, also requires a basic knowledge of programming language translation and static program analysis, as well as run-time components such as memory management.

PL/Functional Programming

Write basic algorithms that avoid assigning to mutable state or considering reference equality.	usage
Write useful functions that take and return other functions.	usage
Compare and contrast (1) the procedural/functional approach (defining a function for each operation with the function body providing a case for each data variant) and (2) the object-oriented approach (defining a class for each data variant with the class definition providing a method for each operation). Understand both as defining a matrix of operations and variants.	not covered
Correctly reason about variables and lexical scope in a program using function closures.	not covered
Use functional encapsulation mechanisms such as closures and modular interfaces.	not covered
Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language.	familiarity

PL/Basic Type Systems

For both a primitive and a compound type, informally describe the values that have that type. [	familiarity
For a language with a static type system, describe the operations that are forbidden statically, such as passing the wrong type of value to a function or method.	familiarity
Describe examples of program errors detected by a type system.	familiarity
For multiple programming languages, identify program properties checked statically and program properties checked dynamically.	not covered
Give an example program that does not type-check in a particular language and yet would have no error if run.	not covered
Use types and type-error messages to write and debug programs.	usage
Explain how typing rules define the set of operations that are legal for a type.	familiarity
Write down the type rules governing the use of a particular compound type.	not covered
Explain why undecidability requires type systems to conservatively approximate program behavior.	not covered
Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections.	not covered
Discuss the differences among generics, subtyping, and overloading.	not covered
Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software.	familiarity