Computer Science Curricula 2023

Version Beta

March 2023

The Joint Task Force on Computing Curricula

Association for Computing Machinery

(ACM)

IEEE-Computer Society

(IEEE-CS)

Association for Advancement of Artificial Intelligence

(AAAI)

Steering Committee members:

ACM members:

- Amruth N. Kumar, Ramapo College of NJ, Mahwah, NJ, USA (ACM CoChair)
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Brett A. Becker, University College Dublin, Ireland
- Richard L. Blumenthal, Regis University, Denver, CO, USA
- Michael Goldweber, Xavier University, Cincinnati, OH, USA
- Pankaj Jalote, Indraprastha Institute of Information Technology, Delhi, India
- Susan Reiser, University of North Carolina Asheville, Asheville, NC, USA
- Titus Winters, Google Inc., New York, NY, USA

IEEE-CS members:

- Rajendra K. Raj, Rochester Institute of Technology (RIT), Rochester, NY, USA (IEEE-CS CoChair)
- Sherif G. Aly, American University in Cairo, Egypt
- Douglas Lea, SUNY Oswego, NY, USA
- Michael Oudshoorn, High Point University, High Point, NC, USA
- Marcelo Pias, Federal University of Rio Grande (FURG), Brazil
- Christian Servin, El Paso Community College, El Paso, TX, USA
- Qiao Xiang, Xiamen University, China

AAAI members:

- Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA
- Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, New York, NY, USA

Table of Contents

CS2023 - The Effort

The History

Several successive curricular guidelines for Computer Science have been published over the years as the discipline has continued to evolve:

- Curriculum 68 [1]: The first curricular guidelines were published by the Association for Computing Machinery (ACM) over 50 years ago.
- Curriculum 78 [2]: The curriculum was revised and presented in terms of core and elective courses.
- Computing Curricula 1991 [3]: The ACM teamed up with the Institute of Electrical and Electronics Engineers – Computer Society (IEEE-CS) for the first time to produce revised curricular guidelines.
- Computing Curricula 2001 [4]: For the first time, the guidelines focused only on Computer Science, with other disciplines such as computer engineering and software engineering being spun off into their own distinct curricular guidelines.
- Computer Science Curriculum 2008 [5]: This was presented as an interim revision of Computing Curricula 2001.
- Computer Science Curricula 2013 [6]: This was the most recent version of the curricula published by the ACM and IEEE-CS.

CS2023 is the next revision of Computer Science curricula. It is a joint effort between the ACM, IEEE-CS, and for the first time, the Association for the Advancement of Artificial Intelligence (AAAI).

The Formation of the Task Force

The **ACM/IEEE-CS/AAAI Task Force** consists of:

- A Steering Committee of 17 members including an ACM co-chair and an IEEE-CS co-chair.
- A committee for each of the 17 knowledge areas

Steering Committee: ACM appointed a co-chair in January 2021. IEEE-CS appointed a co-chair in March 2021. The rest of the Steering Committee was put together as follows:

- Three members were nominated by IEEE-CS co-chair
- Two members were nominated by AAAI
- One member was nominated by ACM CCECC
- The remaining nine members were selected from among the individuals who nominated themselves in response to a Call for Participation posted to multiple ACM SIG mailing lists in February 2021.

The requirements for Steering Committee members were: 1) they were subject experts, 2) willing to work on a volunteer basis, 3) willing to commit at least ten hours a month to CS2023 activities, 4) commit to attending at least two in-person meetings a year; and 5) were aligned with the CS2023 vision. During the Steering Committee formation process in April 2021, the ACM and IEEE-CS co-chairs jointly interviewed each nominee who met the expertise requirements. The candidates who were aligned with the vision and were willing to meet the time commitments were appointed.

Knowledge Area Committees: In June 2021, each Steering Committee member took charge of a knowledge area, and assembled a committee of 5 – 10 subject experts, drawing members from:

- Individuals who had nominated themselves in response to the Call for Participation in February 2021;
- Industry experts; and
- Other Steering Committee members who shared interest in the knowledge area.

Knowledge Area committee members met once a month to discuss curricular revision. While the revision effort was in progress, additional subject experts who expressed interest in volunteering were added to the committees.

The Principles:

Some of the principles that have guided the work of the CS2023 task force include:

- Collaboration: Each knowledge area was revised by a committee of international experts from academia and industry.
- Continuous review and revision: Each version of the curricular draft was anonymously reviewed by at least three outside experts.
- Community engagement: The CS2023 task force reached out to the community through periodic postings to over a dozen ACM Special Interest Group (SIG) mailing lists. It provided multiple channels for the community to participate and contribute. These included conference events such as Birds-of-a-Feather sessions at the ACM SIGCSE Technical Symposium, and email address and feedback form dedicated to each knowledge area, posted on the website csed.acm.org.
- Data-driven change: The early work of CS2023 was informed by surveys of academic institutions and industry practitioners conducted in 2021. Similar surveys are planned for other aspects of the curriculum such as math requirements and core topics.
- Transparency: The work of CS2023 was documented for review and comments by the community on the website: https://csed.acm.org/. Available information included composition of knowledge area committees, results of surveys, and the process used to form the task force.

The Curriculum Revision Process:

The process included the following activities:

- In 2021, surveys were conducted of the current use of CS2013 curricular guidelines and the importance of various components of curricula. The surveys were filled out by 212 academics in the United States, 191 academics from abroad and 865 industry respondents. The summaries of the surveys were incorporated into curricular revision.
- In May 2022, Version Alpha of the curriculum draft was released. It was publicized internationally and feedback solicited. The draft of each knowledge area was sent out to multiple reviewers and at least three reviews were sought for each knowledge area. The reviews were incorporated into the subsequence version of the curricular draft. The draft contained all the knowledge areas from CS2103. For each knowledge area:
    - Knowledge units and topics were listed.
    - Learning outcomes were listed for the topics. The learning outcomes were renamed Illustrative Learning Outcomes to acknowledge that the learning outcomes were at only one or some of all the possible skills levels for each topic.
    - Professional dispositions appropriate for each knowledge area were identified.
    - Computer Science (CS) and Knowledge Area (KA) core topics were identified. These were mostly extensions of Tier I and Tier II topics respectively from CS2013 report.
    - Math needed and wanted was identified as one of two ways in which Math requirements of the major would be identified.
- In March 2023, Version Beta of the curriculum draft was released. This was again sent out for review and posted online for comments and feedback. In Version Beta, the following were incorporated into each Knowledge Area:
    - Revisions based on reviews obtained for Version Alpha
    - Information about packaging each knowledge area into courses
    - An initial competency model with competency specifications for at least authentic tasks identified for the knowledge area
    - A worksheet for core topics, with skill levels associated with each core topic to justify the allocation of core hours to the topic.
- In late summer of 2023, Version Gamma, the pre-lease version of CS2023 will be posted online for a final round of comments and suggestions. In Version Gamma, the following will be incorporated into each knowledge area:
    - Revisions based on reviews obtained for Version Beta
    - Course packaging for the three competency areas: software, systems and applications
    - Curricular packaging
    - Competency specifications that span knowledge areas in each of the three competency areas
- Finally, the report will be released on December 1st, 2023 as follows:
    - A color pdf version, hyperlinked and containing not only curricular recommendations, but also curricular practice articles
    - An online version, hyperlinked and machine-readable
        - Ancillary materials provided online will include revision reports for each knowledge area

- Once endorsed by the ACM Education Board, translations to French, Spanish and Chinese Mandarin – the first version created through auto-translation and the final version edited by a native speaker

References:

1. Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T. E., Keenan, T. A., Kehl, W. B., McCluskey, E. J., Navarro, S. O., Rheinboldt, W. C., Schweppe, E. J., Viavant, W., and Young, D. “Curriculum 68: Recommendations for academic programs in computer science.” *Communications of the ACM*, 11, 3 (1968): 151–197.
2. Austing, R. H., Barnes, B. H., Bonnette, D. T., Engel, G. L., and Stokes, G. “Curriculum ’78: Recommendations for the undergraduate program in computer science.” *Communications of the ACM*, 22, 3 (1979): 147–166.
3. ACM/IEEE-CS Joint Curriculum Task Force. “Computing Curricula 1991.” (New York, USA: ACM Press and IEEE Computer Society Press, 1991).
4. ACM/IEEE-CS Joint Curriculum Task Force. “Computing Curricula 2001 Computer Science.” (New York, USA: ACM Press and IEEE Computer Society Press, 2001).
5. ACM/IEEE-CS Interim Review Task Force. “Computer Science Curriculum 2008: An interim revision of CS 2001.” (New York, USA: ACM Press and IEEE Computer Society Press, 2008).
6. ACM/IEEE-CS Joint Task Force on Computing Curricula. “Computing Science Curricula 2013.” (New York, USA: ACM Press and IEEE Computer Society Press, 2013).

What is new in CS2023?

CS2013 [6] and all the earlier versions of Computer Science curricula (CS2008 [5], CC2001 [4], Curriculum 91 [3], Curriculum 78 [2], Curriculum 68 [1]) presented a knowledge model of Computer Science curricula. The knowledge model consists of:

- **Knowledge areas**, which are silos of related material;
- Each knowledge area is further broken down into **knowledge units**;
- Under each knowledge unit, related **topics** are listed.
- For most topics, learning outcomes are specified.

Currently, a lot of emphasis is being placed on competency models of curricula in various curricular reports, including IT2017 [7], CC2020 [8], Information Systems 2020 [12] and Data Science 2021 [9]. In support of this emphasis, CS2023 includes a competency model. Unlike in previous curricular efforts, CS2023 does not replace knowledge model with competency model, but rather, complements knowledge model with competency model because the two are alternative curricular models, each with its own strengths.

Knowledge Areas: Several knowledge areas were renamed, either to better emphasize their focus or to incorporate changes in the area since 2013:

- Discrete Structures (DS) → Mathematical and Statistical Foundations (MSF)
- Graphics and Visualization (GV) → Graphics and Interactive Techniques (GIT)
- Information Assurance and Security (IAS) → Security (SEC)
- Information Management (IM) → Data Management (DM)
- Intelligent Systems (IS) → Artificial Intelligence (AI)
- Platform Based Development (PBD) → Specialized Platform Development (SPD)
- Programming Languages (PL) → Foundations of Programming Languages (FPL)
- Social Issues and Professional Practice (SP) → Society, Ethics and Professionalism (SEP)

Computational Science was dropped as a knowledge area.

Society, Ethics and Professionalism (SEP):

Given that the work of Computer Science graduates affects all aspects of everyday life, Computer Science as a discipline can no longer ignore or treat as incidental, social, ethical and professional issues. In recognition of this pervasive nature of computing, every other knowledge area in the curricular guidelines now includes a knowledge unit called SEP where topics and learning outcomes at the intersection of the knowledge area and SEP are explicitly listed.

Skill Levels:

CS2013 associated one of the following three **skill levels** with each **learning outcome**:

- Familiarity: “What do you know about this?”
- Usage: “What do you know how to do?”
- Assessment: “Why would you do that?”

In CS2023, four skills levels loosely aligned with revised Bloom’s taxonomy [10] were re-adopted:

Revised Bloom's Taxonomy	**Skill level** with applicable verbs
Remember	**Explain:** define, describe, discuss, enumerate, express, identify, indicate, list, name, select, state, summarize, tabulate, translate
Understand	
Apply	**Apply:** backup, calculate, compute, configure, debug, deploy, experiment, install, iterate, interpret, manipulate, map, measure, patch, predict, provision, randomize, recover, restore, schedule, solve, test, trace, train, virtualize
Analyze	**Evaluate:** analyze, compare, classify, contrast, distinguish, categorize, differentiate, discriminate, order, prioritize, criticize, support, decide, recommend, assess, choose, defend, predict, rank
Evaluate	
Create	**Develop:** combine, compile, compose, construct, create, design, develop, generalize, integrate, modify, organize, plan, produce, rearrange, rewrite, refactor, write

It should be understood that Explain is the pre-requisite skill for the other three levels. The verbs corresponding to each of the four skill levels were adopted from the work of ACM and CCECC []. The purpose of specifying the skill levels has changed:

- Unlike in the past, skill levels were not specified for learning outcomes. Associating skill level with a learning outcome serves merely a redundant descriptive/confirmatory purpose.
- On the other hand, skill levels were specified for all the core topics to justify the estimation of hours associated with the topics. These should be treated as recommended skill levels for core topics.

Professional Dispositions:
The importance of soft skills and personal attributes was included in CS2013 guidelines, albeit in passing. Since then, the importance of developing professional dispositions, defined as *cultivable behaviors desired in the workplace*, has taken on greater significance in computing education and emphasized in competency models of curricula. In CS2023, the most relevant professional dispositions have been listed for each knowledge area. The professional dispositions were picked from the list included in CC2020 [8]. Professional dispositions serve as one of the bridges between knowledge model and competency model of CS2023 curricular guidelines.

Core Hours: In CS2013, core hours were defined along two tiers:

- Tier I: 165 hours
- Tier II: 143 hours

Computer Science programs were expected to cover 100% of Tier I core topics and at least 80% of Tier II topics. While proposing this scheme, CS2013 was mindful that the number of core hours has been steadily increasing in curricular recommendations, from 280 hours in CC2001 to 290 hours in CS2008 and 308 hours in CS2013. Smaller Computer Science programs may find it hard to accommodate the increasing number of core topics in their curricula.

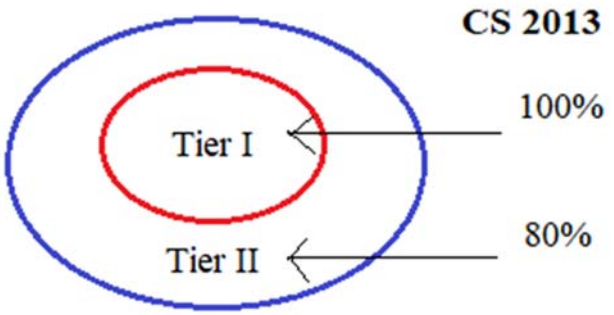


Figure 1: CS2013 Core Topics

In CS2023, a sunflower model of core topics was adopted. In CS2023, core topics are designated as:

- CS core – topics that *every* Computer Science graduate must know
- KA core – topics that *any* coverage of the Knowledge Area *must* include.

This model acknowledges that often, the design of curricula in smaller programs are as much dictated by curricular emphasis dictated by regional needs, the local availability of instructional expertise, and historical evolution of Computer Science programs. While all the programs must cover CS core topics, a program may choose to cover some knowledge areas in greater depth/breadth than other knowledge areas. In figure 2, the highlighted parts reflect the coverage of a typical Computer Science program. Note that the program covers some knowledge areas (KA) in great detail while others in minimal fashion or not at all.

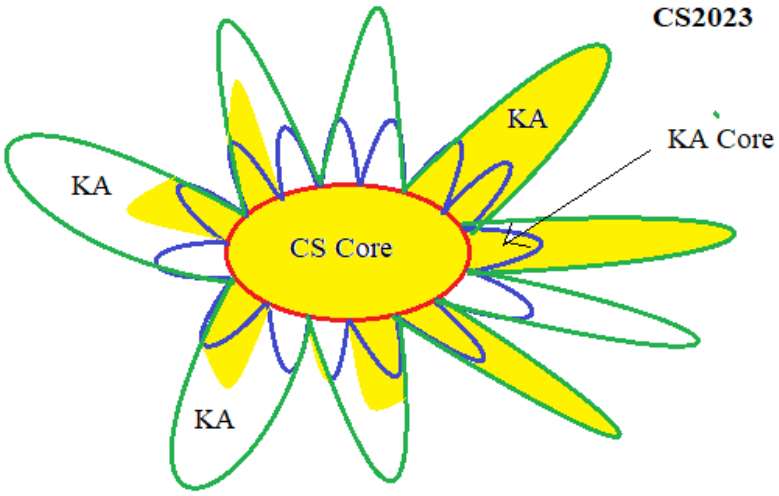


Figure 2: Sunflower model of core topics used in CS2023

Knowledge areas, when chosen coherently, will constitute the competency area of a program. Some competency areas are:

- **Software**, consisting of the knowledge areas Software Development Fundamentals (SDF), Algorithms and Complexity (AL), Foundations of Programming Languages (FPL) and Software Engineering (SE).
- **Systems**, consisting of some of the following knowledge areas: Systems Fundamentals (SF), Architecture and Organization (AR), Operating Systems (OS), Parallel and Distributed Computing (PDC), Networking and Communication (NC), Security (SEC) and Data Management (DM).
- **Applications**, consisting of some of the following knowledge areas: Graphics and Interactive Techniques (GIT), Artificial Intelligence (AI), Specialized Platform Development (SPD), Human-Computer Interaction (HCI), Security (SEC) and Data Management (DM).
- **Theoretical Foundations of Computer Science**, applicable to theoretical exploration of all the knowledge areas, such as pursued in preparation for graduate studies.

Note that software competency area is in part a pre-requisite of the other two competency areas. These competency areas are another bridge between knowledge model and competency model of CS2023 curricular guidelines.

Course Packaging:

A knowledge area is not a course. Many courses may be carved out of one knowledge area and one course may contain topics from multiple knowledge areas. In CS2013, course and curricular examples from various institutions were included in the curricular guidelines. But, adopting such examples from one institution to another would be affected by institutional context, including the level of preparedness of students, the availability of teaching expertise, the availability of pre- and co-requisite courses, etc. Instead, in CS2023, canonical packaging of courses has been provided in terms of knowledge areas and knowledge units.

Competency Model:

Competency is defined as the application of knowledge, skills and professional dispositions in the context of a task []. Note that the knowledge model already encompasses all three ingredients:

- Knowledge in terms of knowledge areas, knowledge units and topics;
- Skills in terms of recommended skill levels for CS and KA core topics
- A list of the most relevant professional dispositions identified for each knowledge area.

Competency model differs from knowledge model in the focus it places on the tasks in which the Computer Science graduate brings to bear all three ingredients. In other words, whereas knowledge model starts with the ingredients and works towards learning outcomes, competency model starts with workplace tasks expected of graduates and works its way backwards to the ingredients needed to complete the tasks.

We proposed a competency model that starts with a list of authentic tasks that demand higher-level skills. In order to identify the tasks, we started with the **competency areas** identified earlier for the KA core topics covered by a program: Software, Systems and Applications. Within each competency area, we identified **competency units**, such as Design, Development, Evaluation, Maintenance, Acceptance, Improvement and Theory. For each competency unit, we

listed one or more tasks and a **competency statement** to complete the task. There could be more than one task per competency unit and more than one competency unit per task. Finally, we listed the knowledge areas, knowledge units and skill levels needed to complete these tasks, thereby bridging between knowledge model and competency model.

Skill levels identified earlier mediate between the knowledge model (knowledge areas, knowledge units and topics) and competency model (competency areas, competency units and tasks), as show in Figure 3. The dashed lines in the figure refer to the need for non-core topics of knowledge units to complete tasks. The links lack directionality to signify that interdependencies work in either direction based on whether one starts with the knowledge model or the competency model. Finally, CS core topics are central to defining Computer Science.

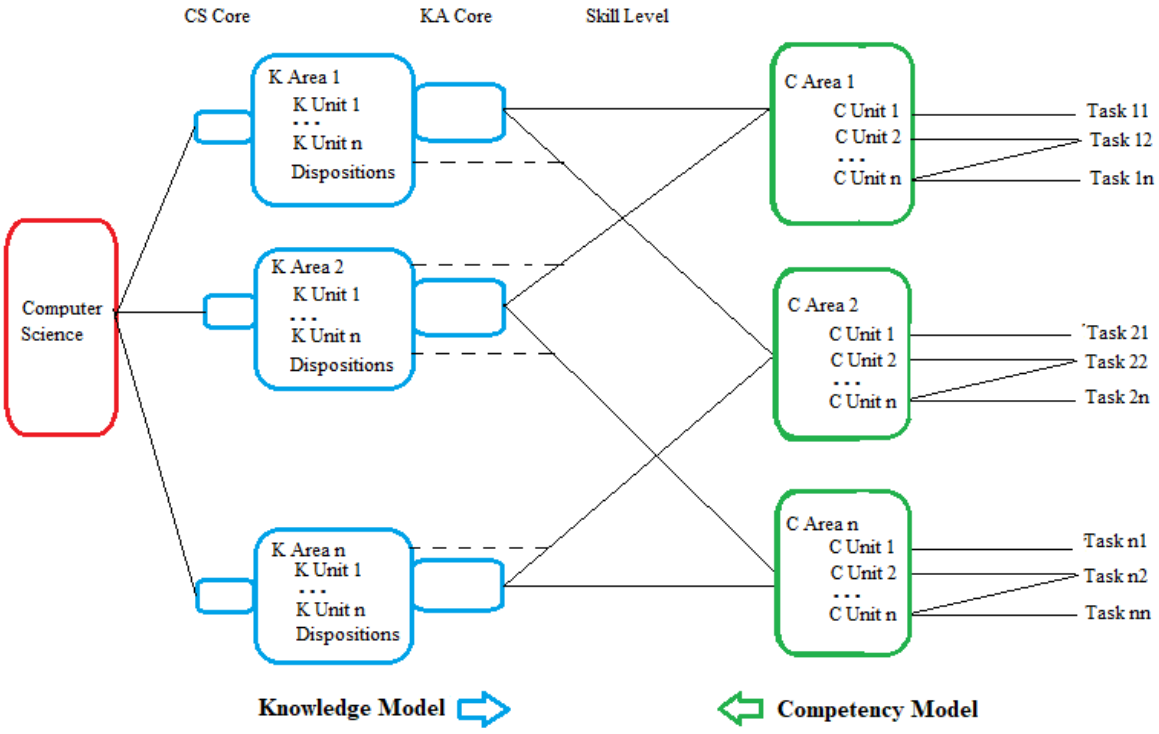


Figure 3: Knowledge Model versus Competency Model

Our competency model differs from earlier attempts [8,11] in many respects:

- We list professional dispositions for knowledge areas and not competency statements. It is not yet well understood how professional dispositions can be fostered among students. It is clear though, that some dispositions are more important at certain stages in a student's development than others, e.g., a novice programmer needs persistence whereas an advanced student needs to be self-directed. So, associating dispositions with knowledge areas makes it easier for the instructor to consider fostering dispositions while bearing the "big picture" in mind. Specifying dispositions for each competency statement often devolves into over-specification.

- We separate the task from the competency statement. Tasks specify what an employer might want done whereas competency statements specify what a graduate might bring to bear in terms of knowledge and skills to attempt the task.
- Instead of exhaustively listing all the topics needed to complete each task, we parsimoniously list the knowledge areas and knowledge units already in place in the knowledge model, thereby inextricably tying the two models together. The knowledge model, by organizing related topics together, helps the instructor teach. The competency model, by focusing on the tasks a graduate will be expected to complete, helps the student apply learning. They complement each other and bring about synergies that neither model can accomplish by itself.

Curricular Practices:

Prior curricular reports enumerated issues in the design and delivery of Computer Science curriculum, e.g., Institutional Challenges section in CS2013. Given the increased importance of many of these issues, in CS2023, it was felt that Computer Science educators should be provided guidelines on incorporating them in their teaching practices. To this end, experts were invited to write in-depth articles about the issues, to be published under the auspices of CS2023. The articles are meant to inform Computer Science educators by providing them the lay of the land without advocating for specific approaches or viewpoints. Some of the topics addressed in these articles include:

- Social issues such as teaching about accessibility, Computer Science for social good, and ethics in Computer Science education.
- Programmatic considerations such as CS + X, the future of Computer Science educational materials and crosscutting themes in Computer Science.
- Professional practices in Computer Science education in settings such as liberal arts and community colleges. In an effort to globalize Computer Science education, articles were also invited on the practices of Computer Science education in various parts of the world. It is hoped that these articles will foster mutual understanding and exchange of ideas, engender transnational collaboration and student exchange, and serve to integrate Computer Science education at the global level through shared understanding of challenges and opportunities.

References

[1] William F. Atchison, Samuel D. Conte, John W. Hamblen, Thomas E. Hull, Thomas A. Keenan, William B. Kehl, Edward J. McCluskey, Silvio O. Navarro, Werner C. Rheinboldt, Earl J. Schweppe, William Viavant, and David M. Young. 1968. Curriculum 68: Recommendations for Academic Programs in Computer Science: A Report of the ACM Curriculum Committee on Computer Science. Communications of the ACM 11, 3 (March 1968), 151–197. https://doi.org/10.1145/362929.362976

[2] Richard H. Austing, Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, and Gordon Stokes. 1979. Curriculum ’78: Recommendations for the Undergraduate Program in Computer Science— a Report of the ACM Curriculum Committee on Computer Science. Communications of the ACM 22, 3 (March 1979), 147–166. https://doi.org/10.1145/359080.359083

[3] Allen B. Tucker, Robert M. Aiken, Keith Barker, Kim B. Bruce, J. Thomas Cain, Susan E. Conry, Gerald L. Engel, Richard G. Epstein, Doris K. Lidtke, Michael C. Mulder, Jean B. Rogers,

Eugene H. Spafford, A. Joe Turner, and Bruce H. Barnes. 1991. Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force. Technical Report. ACM Press and IEEE Computer Society Press, New York, NY, USA.

[4] The Joint Task Force on Computing Curricula. 2001. Computing Curricula 2001. Journal of Educational Resources in Computing. Vol 1 (3) (Sept. 2001), https://doi.org/10.1145/384274.384275

[5] Lillian Cassel, Alan Clements, Gordon Davies, Mark Guzdial, Renée McCauley, Andrew McGettrick, Bob Sloan, Larry Snyder, Paul Tymann, and Bruce W. Weide. 2008. Computer Science Curriculum 2008: An Interim Revision of CS 2001. Technical Report. ACM Press, New York, NY, USA.

[6] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. Computer Science Curricula 2013. Technical Report. ACM Press and IEEE Computer Society Press. https://doi.org/10.1145/2534860 Accessed: November 14, 2017.

[7] Byers, William Newhouse, Bill Paterson, Svetlana Peltsverger, Cara Tang, Gerrit van der Veer, and Barbara Viola. 2017. Information Technology Curricula 2017 (IT2017). Technical Report. Association for Computing Machinery / IEEE Computer Society, New York, NY, USA. https://dl.acm.org/doi/pdf/10.1145/3173161.

[8] Alison Clear, Allen Parrish, John Impagliazzo, Pearl Wang, Paolo Ciancarini, Ernesto Cuadros-Vargas, Stephen Frezza, Judith Gal-Ezer, Arnold Pears, Shingo Takada, Heikki Topi, Gerrit van der Veer, Abhijat Vichare, Les Waguespack, and Ming Zhang. 2020. Computing Curricula 2020 (CC2020): Paradigms for Future Computing Curricula. Technical Report. Association for Computing Machinery / IEEE Computer Society , New York, NY, USA. http://www.cc2020.net/.

[9] Andrea Danyluk and Paul Leidig. 2021. Computing Competencies for Undergraduate Data Science Curricula (DS2021). Technical Report. Association for Computing Machinery, New York, NY, USA. https://www.acm.org/binaries/content/assets/education/curricula-recommendations/dstf_ccdsc2021.pdf.

[10] Anderson, Lorin W.; Krathwohl, David R., eds. (2001). A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives. New York: Longman. ISBN 978-0-8013-1903-7.

[11] Alison Clear, Tony Clear, Abhijat Vichare, Thea Charles, Stephen Frezza, Mirela Gutica, Barry Lunt, Francesco Maiorana, Arnold Pears, Francois Pitt, Charles Riedesel, and Justyna Szynkiewicz. 2020. Designing Computer Science Competency Statements: A Process and Curriculum Model for the 21st Century. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '20). Association for Computing Machinery, New York, NY, USA, 211–246. https://doi.org/10.1145/3437800.3439208

[12] Paul Leidig and Hannu Salmela. 2021. A Competency Model for Undergraduate Programs in Information Systems (IS2020). Technical Report. Association for Computing Machinery, New York, NY, USA. is2020.org

Characteristics of Computer Science Graduates

Some possible characteristics are:

- Algorithmic problem-solver - Good solutions to common problems at an appropriate level of abstraction
- Competent programmer
- In possession of mental model of computation - deep learning
- Life-long learner
- Collaborative
- Socially responsible - ethical behavior
- Global and cultural competence
- Cross-disciplinary - understanding of non-computing disciplines
- Adversarial thinker/Computational thinker
- Think at multiple levels of abstraction.
- Adaptable
- Handle ambiguity and uncertainty
- Analytical and problem-solving skills
- Knowledge of algorithms and data structures
- Familiarity with software engineering principles
- Strong mathematical and logical skills
- Effective communication skills

Institutional Challenges

Some institutional challenges are:

- Faculty recruitment and retention
- Workload to manage explosive enrollments
- AI generation of code and its implications
- Integrating AI into the core educational requirements
- Fragmentation (Data Science, AI, etc. etc.) are all becoming degrees in their own right
- Rethinking core curriculum
- Unprepared high school students - Getting students to put the effort in to be successful as CS students.
- Elevating teaching track to professional status
- Certificates, micro-credentials, associates in computing
- CS for all (packaging it in an accessible way to a broad set of majors)
- BPC
- Online education

Algorithmic Foundations (AL)

Preamble

Algorithms and data structures are fundamental to computer science and software engineering since every theoretical computation and real-world program consists of algorithms that operate on data elements possessing an underlying structure. Selecting appropriate computational solutions to real-world problems requires understanding the theoretical and practical capabilities and limitations of available algorithms including their impact on the environment and society. Moreover, this understanding provides insight into the intrinsic nature of computational problems as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or other implementation aspects.

This area focuses on the nature of algorithmic computation including the concepts and skills required to design, implement, and analyze algorithms for solving real-world computational problems. As algorithms and data structures are essential in all advanced areas of computer science, this area provides the foundations that every computer science graduate is expected to know. There is a complementary relationship among many topics in this area and the Software Development Fundamentals (SDF) area allowing them to be simultaneously presented.

Changes since CS 2013: This area has been renamed since the CS'13 curricular guidelines to better reflect the foundational scope of the included concepts. It also returns to early CS curricular guidelines that recommended required exposure to foundational computational theory concepts. To reinforce the important impact of computation on society, an algorithms and society unit is also now included.

Core Hours

Knowledge Unit	CS Core	KA Core
Fundamental Data Structures and Algorithms	36	7
Algorithmic Strategies	4	
Complexity Analysis	4	2
Computational Models	9	23
Algorithms and Society	4	
Total	**57**	**32**

Knowledge Units

AL/Fundamental Data Structures and Algorithms

[24 CS Core hours, 7 KA Core Hours]

This unit focuses on the data structures and algorithms every computer science graduate is expected to know. Topics in this unit complement those in the Software Development Fundamentals (SDF) and Math Foundations (MF) areas, and the AL/Algorithmic Strategies unit.

Topics [CS Core]:

- Abstract Data Types (ADTs) including Bag, Collection, Dictionary, List, and Set properties [*cross-reference* SDF/Fundamental Data Structures]
- Arrays: single and multi-dimensional
  - Linear and Binary Search
- Cryptography (e.g. SHA-256, RSA) [*crosslist*: SEC/Cryptography]
- Graphs [*cross-reference*: MF/Graphs and Trees]
  - (un)directed, (a)cyclic, (un)connected, and (un)weighted
  - Adjacency List and Matrix representations
- Graph Algorithms
  - Breadth-First Search
    - Connectivity, Shortest-Path
  - Depth-First Search
    - Acyclicity, Connectivity, Transitive Closure, Topological Sort
  - Hamiltonian Circuit
  - Minimal Spanning Tree
    - Prim's and Kruskal's algorithms
  - Shortest-Path
    - Bellman-Ford,
    - Dijkstrat's/Uniform-Cost,
    - Floyd-Warshall
  - Transitive Closure: Warshall's Algorithm
- Hash Tables / Maps
  - Collision resolution: Linear/Quadratic Probing, Chaining, and Rehashing
- Linked Lists: single, doubly linked, and circular
- Objects [*cross-reference* PL/Object-Oriented Programming]
- Queues, Priority Queues, and Dequeues
- Records/Structs and Tuples
- Stacks
- Strings
  - String Matching: (Boyer-Moore),
- Sorting Algorithms:
  - $O(n^2)$ Selection, Insertion
  - $O(n \log n)$ Quicksort, Merge, and Heap
  - $O(n)$ Fast-Sorting: Bucket and Radix

    - Lexicographical Ordering
    - Partial Ordering
    - Topological Ordering
- Trees
    - Binary, N-ary, Search, Balanced, and Heaps
    - Depth-First, Bread-First, Best-First, Backtracking search
    - Balancing approaches (e.g., for AVL, Red-Black, 2-3, or B trees)
    - Huffman Coding
- Differential Privacy
- Basic Linear Algebra (e.g., Strassen's Matrix Multiplication)
- Invariants (in: loops, search algorithms, etc.)

Topics [KA Core]

- Consensus Algorithms (e.g. Blockchain)
- Geometric Algorithms (e.g., Graham Scan for Convex Hull)
- Linear Programming (e.g., Simplex algorithm)
- Approximation algorithms (e.g., for Knapsack, Traveling Salesperson)
- Randomized Algorithms (e.g. MaxCut, Balls and Bins)
- Computational Geometry (e.g. Closest Pair, Convex Hull)
- Branch and Bound

Topics [Elective]

- Quantum Algorithms (e.g., Deutsch-Jozsa, Bernstein-Vazirani, Simon's, Shor's, Grover's)
- Signal Processing (e.g. Fast Fourier Transform and Convolution)
- Cryptographic Hashing (e.g. Rabin's Algorithm (Digital Fingerprint)
- Evolutionary Algorithms (e.g., Genetic Algorithms) [*crosslist*: Artificial Intelligence]

Illustrative Learning Outcomes and Competencies [CS Core]

1. For each Fundamental ADT/Data Structure in this unit:
    a. Articulate its definition, properties, representation(s), and operations it supports,
    b. Use a real-world example, explain step-by-step how the operations associated with the data structure transform it.
2. For each of the algorithms in this unit,
    a. Use a real-world example, show step-by-step how the algorithm operates.
3. Given a problem that involves multiple actions, such as insertions, deletions, and searches, that may or may not have predictable occurrences, properties, or relationships, devise a data structure that optimally supports such actions.
4. Given requirements for a real-world application, create multiple design solutions using various data structures and algorithms. Subsequently, evaluate the suitability, strengths, and weaknesses of the selected approach for satisfying the requirements.
5. Prepare a presentation that justifies the selection of appropriate data structures and/or algorithms to solve a given real-world industry problem.

AL/Algorithmic Strategies

[4 CS Core hours, Elective Hours]

This unit focuses on the fundamental design strategies used in algorithmic problem-solving. The small number of allocated hours reflect the fact that the listed representative examples of these strategies are examined in the AL/Fundamental Data Structures and Algorithms knowledge unit.

Topics [CS Core]:

- Algorithmic Strategy
- Approximation [*cross-reference* AL/Complexity Analysis]
    - Polynomial approximation
- Backtracking [*crosslist*: Artificial Intelligence]
- Branch and Bound [*cross-reference* Artificial Intelligence]
- Brute-Force/Exhaustive Search
    - Selection Sort, Traveling Salesman, Knapsack
- Consensus algorithms [*cross-reference* SEC/Cryptography]
    - Blockchain
- Decrease-and-Conquer
    - Insertion sort, Depth and Breadth-First search, Topological sort
- Divide-and-Conquer
    - Binary Search, Quicksort, Mergesort, Strassen's algorithm
- Dynamic Programming
    - Warshall's algorithm and Floyd's algorithm
- Greedy
    - Dijkstra's Kruskal's algorithms
- Heuristic: A* [*cross-reference* Artificial Intelligence]
- Iterative
    - Linear Search
- Parallel [*crosslist*: PD/Parallel Algorithms, Analysis, and Programming]
    - Parallel Mergesort
- Randomized/Stochastic Algorithms
    - MaxCut, Balls and Bins
- Recursive
    - Depth-First, Breadth-First Search, Factorial
- Space and Time Tradeoff
    - Hashing
- Transform-and-Conquer/Reduction
    - 2-3, AVL, Red-Black trees, Heapsort

Illustrative Learning Outcomes and Competencies [CS Core]

1. For each of the algorithms in the AL/Fundamental Data Structures and Algorithms unit, identify the algorithmic strategy it uses and articulate how the algorithm exemplifies the corresponding strategy demonstrating how it solves a specific problem instance:
    a. How binary search and mergesort, quicksort, and insertion sort, as well as Strassen's Algorithm, exemplify a divide-and-conquer algorithmic strategy by using them to find an element in or sort an unsorted array of elements.

    b. How topological sorting using depth-first search exemplifies a decrease-and-conquer algorithmic strategy
    c. How Dijkstra's and Kruskal's algorithm exemplify a greedy algorithmic strategy by solving a single-source shortest-path graph problem.
    d. How Warshall's Algorithm exemplifies a dynamic programming algorithmic strategy by using it to find the transitive closure of a directed graph.
    e. How Min-max with Alpha-Beta pruning exemplifies heuristic, branch-and-bound, and back-tracking strategies.
    f. How pre-sorting a list exemplifies a transform-and-conquer strategy.
    g. How blockchain, as used by crypto-currency applications (e.g. Bitcoin) exemplifies a consensus algorithmic strategy.
2. Transform between common recursive and iterative problem-solving algorithms.
3. Determine an appropriate algorithmic approach to an industry problem and use appropriate strategies (e.g. brute-force, greedy, divide-and-conquer, recursive backtracking, dynamic program, etc.) that considers the tradeoffs among these strategies

AL/Complexity Analysis

[4 CS Core hours, 2 KA hours]

This unit focuses on analyzing the time and space efficiency of algorithms including the definition of tractability. The small number of allocated hours reflects the fact that algorithms demonstrating the various complexity classes are examined in the AL/Fundamental Data Structures and Algorithms knowledge unit.

Topics [CS Core]

- Algorithmic Analysis
- Analysis Framework
    - Average, Best, and Worst case performance
    - Empirical and Relative (Order of Growth) Measurements
    - Input Size and Primitive Operations
    - Time and Space Efficiency
- Asymptotic complexity analysis
    - Big O, Little O, Big Omega, and Big Theta
    - Foundational Complexity/Efficiency classes
        - Constant, Logarithmic, Linear, Log Linear, Quadratic, Cubic, Exponential, and Factorial
- Iterative and recursive algorithm analysis
    - Recurrence Relations
        - Master Theorem Analysis
        - Substitution Analysis
- Tractability and Intractability
    - P, NP and NP-C complexity classes
        - Hamiltonian Circuit, Knapsack, and SAT problems
- Time and space trade-off in algorithms

Topics [KA Core]

- Turing Machine-Based Models of Complexity
    - P, NP, and NP-C complexity classes
- Space Complexity: Savitch's Theorem, NSpace, PSpace

Illustrative Learning Outcomes and Competencies [CS Core]

1. Prepare a presentation that informally introduces non-technical managers to the basic concepts of algorithmic complexity that formally uses Big O, Omega, and Theta. Asymptotic notation to describe the efficiency of representative algorithms of each fundamental complexity class.
2. Demonstrate the complexity of each algorithm listed in the AL/Fundamental Data Structures and Algorithms unit.
3. Perform empirical studies to determine and validate hypotheses about the runtime complexity of various algorithms.
4. Give examples that illustrate time-space trade-offs of algorithms.
5. Use recurrence relations to determine the time complexity of recursively defined algorithms by solving elementary recurrence relations (e.g., Substitution and the Master Theorem) and present the results to a group of scholars.
6. Explain the significance of P, NP, and NP-Completeness within a business context.

Illustrative Learning Outcomes and Competencies [KA Core]

7. Explain how approximation algorithms are used (e.g. "solve" Traveling Salesperson).

AL/Computational Models

[9 CS Core hours, 21 KA Hours]

This unit focuses on the definitions, capabilities, and limitations of common models of computation including the equivalent formalizations of algorithmic computation by a computer. The relation of computation to formal languages and computable functions is also examined. The models and languages in this unit complement topics in the DS/Basic Logic, PD/Parallel and Distributed Computing, and PL/Programming Languages knowledge areas.

Topics [CS Core]:

- Formal Languages and Grammars
    - Chomsky Hierarchy
    - Regular, Context-Free, Context-Sensitive, and Recursively Enumerable
    - Regular expressions
- Formal Automata
    - Finite State, Pushdown, Linear-Bounded, and Turing Machine
    - Deterministic versus Nondeterministic and equivalencies
    - Relations among formal automata, languages and grammars
    - Decidability and limitations
- Decidability, Computability, Halting problem
- The Church-Turing Thesis

- The P, NP, and NPC complexity [*crosslist*: AL/Complexity Analysis]

Topics [KA Core]:

- Equivalent Models of Computation
  - Turing Machine variations (e.g. multi-tape, non-deterministic)
  - Lambda Calculus and Mu-Recursive Functions
- Pumping Lemmas (Finite State and Pushdown Automata)
- Decidability (Arithmetization and Diagonalization)
- Reducibility
- Time Complexity based on Turing Machine
- Space Complexity (e.g. PSPACE, Savitch's Theorem)
- Quantum Computing
- Concurrent Systems (e.g., non-termination, non-determinism, "thread" interference)

Illustrative Learning Outcomes and Competencies [CS Core]

1. Compare and contrast the definitions and computational capabilities/limitations of: Finite State, Pushdown, Linear-Bounded, and Turing Machine automata.
2. Design an abstract machine that accepts a specified language.
3. Design an automaton for an organization that represents a given process.
4. Design a Regular Expression to represent a specified language.
5. Compare and contrast Regular, Context-free, Context sensitive, and Recursively Enumerable/Computable (Chomsky Type-X) languages and their uses.
6. Design a Regular or Context-free grammar to represent a specified language.
7. Explain the Halting problem and its significance.
8. Explain the Church-Turing Thesis and its significance.
9. Explain the P, NP, and NPC complexity classes and their significance.

Illustrative Learning Outcomes [CS Core]

1. Use a pumping lemma to prove the limitations of Finite State and Pushdown automata.
2. Use arithmetization and diagonalization to prove Turing Machine Halting/Undecidability.
3. Explain a reduction such as between Halting and Undecidability of the language accepted by a Turing Machine, where one has been proven by diagonalization.
4. Explain the Primitive Recursive functions (zero, successor, selection, primitive recursion, and composition) and their significance.
5. Explain the General Recursive functions and the significance of the Mu-Operator.
6. Design a Turing Machines that compute the Primitive and General Recursive Functions
7. Explain how computation is performed in Lambda Calculus.
8. For a quantum system give examples that explain the following postulates:
   a. State Space: system state represented as a unit vector in Hilbert space,
   b. State Evolution: the use of unitary operators to evolve system state,
   c. State Composition: the use of tensor product to compose systems states,
   d. State Measurement: the probabilistic output of measuring a system state.
9. Explain the significance of non-termination, non-determinism, and interference in concurrent models of computation.

AL/Algorithms and Society
[4 CS Core hours]
This unit focuses on topics related to the impact of algorithmic computation on all aspects of society, recognizing that graduates have an obligation to use computation to benefit society.

Topics [CS Core]

- Context-Aware Computing [*crosslist*: SEP/Social Context]
- Social, Ethical, and Secure Algorithms
- Differential Privacy
- Algorithmic Fairness [*crosslist*: AI?]
- Accountability/Transparency

Illustrative Learning Outcomes [CS Core]

1. Devise algorithmic solutions to real-world societal problems, such as routing an ambulance to a hospital
2. Predict and explain the impact that an algorithm may have on the environment and society when used to solve real-world problems taking into account that it can affect different societal groups in different ways.
3. Articulate how differential privacy protects knowledge of an individual's data.
4. Articulate how Public Key encryption protects knowledge of secure data.
5. Articulate the pros and cons of using blockchain algorithms to support crypto-currency (e.g. as used by Bitcoin).
6. Write a white paper assessing the social and ethical appropriateness of an algorithmic solution to a business problem.

Professional Dispositions

Overarching is a growth-mindset

- A curiosity for exploring how software solves real-world problems.
- An enthusiasm for determining efficient software solutions to real-world problems
- Self-confidence in one's ability to find algorithmic solutions to real-world problems
- An appreciation for the impact of CS theory on solving real-world problems.

Math Requirements

Required:

- MSF/Sets, Relations, Functions
- MSF/Basics of Counting
    - Set cardinality and counting
    - Solving recurrence relations
- MSF/Graphs and Trees
- MSF/Proof Techniques

Shared Concepts and Crosscutting Themes

Shared concepts:

- Software Development Fundamentals
  - SDF/Fundamental Data Structures
  - SDF/Algorithms and Design
- Programming Languages
  - PL/Object-Oriented Programming
  - PL/Syntax Analysis
- Information Assurance and Security
  - IAS/Cryptography
- Artificial Intelligence
  - AI/Basic Search Strategies

Competency Specifications

- **Task 1:** Appropriately route an ambulance from a location to the nearest hospital.
- **Competency Statement:** Research whether there exists a known algorithm and associated data structure(s) that efficiently solves a given computational problem.
- **Competency area :** Software
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units:**
  - AL / Fundamental Data Structures and Algorithms
  - AL / Algorithmic Strategies
  - AL / Complexity Analysis
- **Required skill level:** Evaluate
- **Core level:** CS core

- **Task 2:** Given a set of preference characteristics input by a user, prioritize a set of recommended charities to which they may wish to donate.
- **Competency Statement:** Design an efficient algorithm and associated data structure(s), perhaps by mixing known algorithms, to resolve a computational problem.
- **Competency area:** Software
- **Competency unit:** Develop
- **Required knowledge areas and knowledge units:**
  - AL / Fundamental Data Structures and Algorithms
  - AL / Algorithmic Strategies
  - AL / Complexity Analysis
- **Required skill level:** Develop

- **Core level:** CS core

- **Task 3:** Redesign a legacy customer service application to be more efficient.
- **Competency Statement:** Given an algorithmic solution to a computational problem, determine whether a more efficient solution to the problem exists
- **Competency area:** Software
- **Competency unit:** Design / Improvement
- **Required knowledge areas and knowledge units:**
    - AL / Fundamental Data Structures and Algorithms
    - SDF / Fundamental Data Structures and Algorithms
    - AL / Complexity Analysis
- **Required skill level:** Develop
- **Core level:** CS core

- **Task 4:** Design an algorithm that tracks the status of a patient scheduled for a medical procedure from initial check-in through discharge including follow-up visits.
- **Competency Statement:** If appropriate, design a finite state or pushdown automata that solves a given computational problem.
- **Competency area:** Software
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Integration / **Documentation** / **Evaluation** / Maintenance / Management / Consumer Acceptance / Adaptation to social issues / Improvement
- **Required knowledge areas and knowledge units:**
    - AL / Algorithms and Society
    - SEP / Social Context
- **Required skill level:** Explain / Apply / Evaluate / Develop
- **Core level:** CS core

- **Task 5:** Given a program or proposed program that satisfies a set of requirements
- **Competency Statement:** Write a white paper explaining why an algorithm is an appropriate choice for resolving a computation problem. [CS Core]
- **Competency area:** Theory
- **Competency unit:** Documentation / Evaluation
- **Required knowledge areas and knowledge units:**
    -
- **Required skill level:** Explain
- **Core level:** CS core

- **Task 6:** Create a scheduling application for a business that handles multiple: tasks, due dates, resources, and performance rewards/penalties.
- **Competency Statement:** Write a white paper explaining why the algorithmic solution to a computation problem is intractable. [CS Core]
- **Competency area:** Theory
- **Competency unit:** Design / Development
- **Required knowledge areas and knowledge units:**
    - AL / Computational Models
- **Required skill level:** Explain / Apply
- **Core level:** CS core

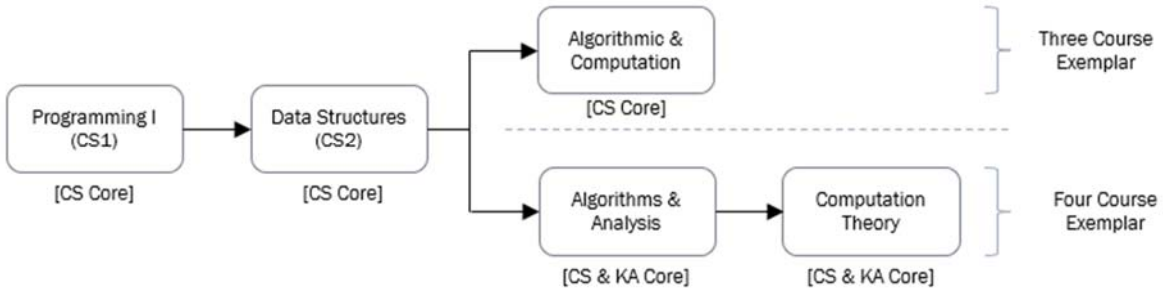
- **Task 7:** Given the business problem of saving money in the storage and transmission of data, create a single algorithmic solution that determines the optimal amount that any given business data can be compressed without losing information.
  [Note: this task reduces to the halting problem.]
- **Competency Statement:** Write a white paper explaining why the algorithmic solution to a computation problem is undecidable. [CS Core]
- **Competency area:** Theory
- **Competency unit:** Design / Documentation
- **Required knowledge areas and knowledge units:**
    - AL / Algorithms and Society
    - SEP / Social Context
- **Required skill level:** Explain / Evaluate
- **Core level:** CS core

- **Task 8:** Design an algorithm that rates people's suggestions according to how well the suggestion is liked.
- **Competency Statement:** Write a white paper explaining the potential impact an algorithmic solution to a computation problem might have on society and the environment. [CS Core]
- **Competency area:** Software
- **Competency unit:** Adaptation to social issues
- **Required knowledge areas and knowledge units:**
    - AL / Algorithms and Society
    - SEP / Social Context
- **Required skill level:** Explain
- **Core level:** CS core

- **Task 9:** Design an algorithm that automates a manual task.
- **Competency Statement:** Write a white paper explaining the potential impact an algorithmic solution to a computation problem might have on society and the environment. [CS Core]
- **Competency area:** Software
- **Competency unit:** Adaptation to social issues / Improvement
- **Required knowledge areas and knowledge units:**
    - AL / Algorithms and Society
    - SEP / Social Context
- **Required skill level:** Explain / Apply / Evaluate / Develop
- **Core level:** CS core

Course Packaging Suggestions

As depicted in the following figure, the committee envisions two common approaches for addressing AL topics in CS courses. Both approaches included required introductory CS1 and CS2 programming courses. The three-course approach, all of the CS Core topics are covered since it includes an algorithms course that is primarily focused on algorithms and complexity, but includes the CS Core topics on computation theory. In the four-course approach, theory of computation topics are addressed in a related course leaving room to address additional topics in the Algorithms & Analysis course. It is also possible that topics from the AL area may also be (further) addressed in more advanced courses, such as finite state and pushdown topics in a Programming Languages/compiler course or backtracking and branch-and-bound algorithms in an Artificial Intelligence focused course.



Courses Common to CS and KA Core Approaches

CS1: Programming I

KU	Topics	Hours
Fundamental Data Structures and Algorithms	Arrays and strings including Linear Search Objects: Simple object definition	1 1
Algorithmic Strategies	Iterative, Brute Force	*

Complexity Analysis	Best, average, and worst case algorithmic behavior	*

*Insignificant (i.e., ten minutes or less introducing the topic and its concepts)

CS2: Data Structures

KU	Topics	Hours
Fundamental Data Structures	Abstract Data Types (ADTs), Bag, Collection, Dictionary Lists, and Set properties Arrays: Binary Search and Multi-dimensional Linear Search Hash Tables/Maps including conflict resolution strategies Linked Lists - single and double Objects: interfaces, inheritance Queues, Dequeues, and Stacks Trees: Binary, Ordered, Breadth- and Depth-first search An $O(n^2)$ sort, such as Selection Sort An $O(n \log n)$ sort, such as Quicksort Secure Hash Algorithm 256 (SHA-256)	15
Algorithmic Strategies	Recursive (factorial example, tree search) Divide-and-Conquer	1 *
Complexity Analysis	Best, worst, average case algorithmic behavior Upper asymptotic Bounds	*
Algorithms and Society		2

*Insignificant (i.e., ten minutes or less introducing the topic and its concepts)

Single Course CS Core Only Approach

Algorithms & Computation

KU	Topics	Hours
Fundamental Data Structures	Insertion Sort and Merge Sort $O(n)$ Fast-Sorting: Bucket and Radix, Lexical ordering Graphs (un)directed, (a)cyclic, (un)connected, and (un)weighted Adjacency List and Matrix representations Shortest-path: Dijkstra's/Uniform-Cost, Bellman-Ford, Floyd-Warshall Transitive Closure: Warshall's Algorithm, Depth-First Minimal Spanning Tree: Prim's and Kruskal's algorithms Topological Sort	16

	Priority queues, Heaps, and Heap sort String Matching (Boyer-Moore), Trees: Search, Balanced, such as AVL, Huffman Coding Matrix Multiplication P, NP, NPC (Traveling salesperson, Knapsack) Best-First Search, Backtracking, Branch and bound	
Algorithmic Strategies	All CS Core	2
Complexity Analysis	All CS Core	3
Computational Models	All CS Core	9
Algorithms and Society	Differential Privacy	2

*I - Insignificant (i.e., a few minutes introducing the topic and its concepts)

Two Courses for CS and KA Core Approach

Algorithms & Analysis

KU	**Topics**	**Hours**
Fundamental Data Structures	Insertion Sort and Merge Sort $O(n)$ Fast-Sorting: Bucket and Radix, Lexical ordering Graphs (un)directed, (a)cyclic, (un)connected, and (un)weighted Adjacency List and Matrix representations Shortest-path: Dijkstra's/Uniform-Cost, Bellman-Ford, Floyd-Warshall Transitive Closure: Warshall's Algorithm, Depth-First Minimal Spanning Tree: Prim's and Kruskal's algorithms Topological Sort Priority queues, Heaps, and Heap sort String Matching (Boyer-Moore), Trees: Search, Balanced, such as AVL, Huffman Coding Matrix Multiplication P, NP, NPC (Traveling salesperson, Knapsack) Best-First Search, Backtracking, Branch and bound	22
Algorithmic Strategies	All CS Core	2
Complexity Analysis	All CS Core	5
Algorithms and Society	Differential Privacy	2

*I - Insignificant (i.e., a few minutes introducing the topic and its concepts)

Computational Theory

KU	Topics	Hours
Computational Models	All CS and KA Core	32
Algorithms and Society		*

*I - Insignificant (i.e., a few minutes introducing the topic and its concepts)

Committee

Chair: Richard Blumenthal, Regis University, Denver, USA

Members:

- Cathy Bareiss, Bethel University, Mishawaka, USA
- Tom Blanchet, Hillman Companies Inc., Boulder, USA
- Doug Lea, State University of New York at Oswego, Oswego, USA
- Sara Miner More, John Hopkins University, Baltimore, USA
- Mia Minnes, University of California San Diego, USA
- Atri Rudra, University at Buffalo, Buffalo, USA
- Christian Servin, El Paso Community College, El Paso, USA

Appendix: Core Topics and Skill Levels

The increase in core hours is partially motivated in that virtually all existing CS programs have a CS2 Data Structures and subsequent Algorithms course giving 64 total hours of available lecture time. Furthermore, curricular guidelines, such as those for Liberal Arts colleges, support the inclusion of algorithms.

KA	KU	Topic	Performance	CS/KA	Hours
AL	Algorithmic Strategies	Algorithmic Strategy ● Approximation [AL/Complexity Analysis] ● Backtracking [*crosslist*: Artificial Intelligence] ● Branch and Bound [Artificial Intelligence] ● Brute-Force/Exhaustive Search ● Consensus algorithms [SEC/Cryptography] ● Decrease-and-Conquer ● Divide-and-Conquer ● Dynamic Programming	Explain	CS	4

		• Greedy • Heuristic: A* [*cross-reference* AI] • Iterative • Parallel [*crosslist*: PD/Parallel Algorithms, Analysis, and Programming] • Randomized/Stochastic Algorithms • Recursive • Space and Time Tradeoff • Transform-and-Conquer/Reduction			
		• Algorithmic Strategy (example algorithms) [AL/Fundamental Algorithms and Data Structures]	Solve		—

AL	Complexity Analysis	• Analysis Framework
		  ○ Average, Best, and Worst case performance
		  ○ Empirical and Relative (Order of Growth) Measurements
		  ○ Input Size and Primitive Operations
		  ○ Time and Space Efficiency and tradeoffs
		• Asymptotic complexity analysis
		  ○ Big O, Little O, Big Omega, and Big Theta
		  ○ Foundational Complexity/Efficiency classes
		  ■ Constant, Logarithmic, Linear, Log Linear, Quadratic, Cubic, Exponential, and Factorial
		• Iterative and recursive algorithm analysis
		  ○ Recurrence Relations
		  ■ Master Theorem Analysis
		  ■ Substitution Analysis
		• Tractability and Intractability
		  ○ P, NP and NP-C complexity classes
		  ■ Hamiltonian Circuit, Knapsack, and SAT problems
			Explain	CS	1
					
					1
					
					1
					
					1

		• Algorithmic Analysis (using) [AL / Fundamental Algs and Data Structures}	Solve		—

		• Turing Machine-Based Models of Complexity
		  ○ P, NP, and NPC complexity
		• Space Complexity
		  ○ NSpace, PSpace, Savitch's Theorem
			Explain	KA	2

AL	Fundamental Algorithms & Data Structs (FA/DS)	• Abstract Data Types (ADTs) including Bag, Collection, Dictionary, List, and Set properties [SDF/Fundamental Data Structures]	Explain	CS	1
	FA/DS Strategies Complexity	• Arrays: single and multi-dimensional ○ Linear and Binary Search • Brute Force, Divide and Conquer • Complexity Class: Linear, Logarithmic	Explain Solve Explain Explain		2 — —

	FA/DS	● Cryptography: AES (SHA-256), RSA	Explain		2
	FA/DS	● Graphs [MF/Graphs and Trees] ○ (un)directed, (a)cyclic, (un)connected, and (un)weighted ○ Adjacency List and Matrix representations ● Graph Algorithms ○ Breadth-First Search ■ Connectivity, Shortest-Path ○ Depth-First Search ■ Acyclicity, Connectivity, Topological Sort, Transitive Closure ○ Hamiltonian Circuit ○ Minimal Spanning Tree ■ Prim's and Kruskal's algorithms ○ Shortest-path ■ Bellman-Ford ■ Dijkstra's/Uniform-Cost ■ Floyd-Warshall ○ Transitive Closure ■ Warshall's Algorithm	Explain Solve		1 1 1 1 1 1 1 1 1 1 1
	Strategies Complexity	● Dynamic Programming, Greedy ● Complexity classes: Quadratic, Exponential			— —
	FA/DS	● Hash Tables / Maps ○ Collision resolution: Linear/Quadratic Probing, Chaining, and Rehashing	Explain		2
	Strategies Complexity	● Space and Time Tradeoffs ● Complexity Class: Constant O(1)			— —
	FA/DS Strategies Complexity	● Linked Lists: single, doubly linked, and circular ● Brute-Force ● Complexity classes:	Solve		1 — —
	FA/DS	● Objects [PL/Object-Oriented Programming]			—
	FA/DS	● Queues, Priority Queues, and Dequeues			1
	FA/DS	● Records/Structs and Tuples			—
	FA/DS	● Stacks			1
	FA/DS	● Strings ○ String Matching (Boyer-Moore),			1
	FA/DS	● Trees: Binary, n-ary, Search ○ Depth-First, Breadth-First, Best-First, and Backtracking search ○ Balanced ■ 2-3, AVL, B, or Red-Black,	Solve		3 1

	Strategies Complexity	■ Heap and Heapsort ○ Huffman Coding ● Divide-and-Conquer, Greedy, Transform-and-Conquer ● Complexity Classes: Linear, Logarithmic			1 1 — —
	FA/DS Strategies Complexity	● Sorting Algorithms: ○ $O(n^2)$ Selection, Insertion ○ $O(n \log n)$ Quicksort, Merge, and Heap ○ $O(n)$ Fast-Sorting: Bucket and Radix ○ Lexicographical Ordering ○ Partial Ordering ● Brute Force, Divide-and-Conquer ● Complexity classes: Quadratic	Solve		 1 2 1 — —
	FA/DS Strategies Complexity	● Basic Linear Algebra ○ Matrix Multiplication ■ Strassen's Matrix Multiplication ● Divide-and-Conquer ● Complexity Classes: Quadratic, Cubic			1
	FA/DS	● Differential Privacy	Explain		1
	FA/DS	● Invariants (loop, in search algorithms)	Explain		2
AL	Fundamental Data Structures and Algorithms	● Consensus Algorithms (e.g. Blockchain) ● Geometric Algorithms (e.g., Graham Scan for Convex Hull) ● Linear Programming (e.g., Simplex algorithm) ● Approximation algorithms (e.g., for Knapsack, Traveling Salesperson) ● Randomized Algorithms (e.g. MaxCut, Balls and Bins)	Explain	KA	7
AL	Complexity Analysis	● Turing Machine-Based Models of Complexity ○ P, NP, and NPC complexity ● Space Complexity ○ NSpace, PSpace, Savitch's Theorem	Explain	KA	2
AL	Automata and Computability	● Formal Automata, Languages and Grammars, Chomsky Hierarchy ○ Deterministic versus Nondeterministic equivalencies ○ Relations among formal automata, languages, and grammars ● Finite State, Regular Grammar/Language ○ Regular Expressions ● Pushdown, Context-Free Grammar/Languages ● Turing Machine, Recursively Enumerable ○ Linear-Bounded, Context-Sensitive ● Decidability, Computability, Halting problem The Church-Turing Thesis	Explain	CS	2 1 1 1 2 1

		● Finite State, Regular Grammars & Expressions ● Pushdown, Context-Free Grammars	Solve		 1
	Automata and Computability	● Equivalent Models of Computation ○ Turing Machine variations ○ Lambda Calculus ○ Mu-Recursive Functions ● Pumping Lemma Proofs ○ Finite State and Pushdown Automata ● Decidability ○ Arithmetization ○ Diagonalization ● Reducibility ● Time Complexity based on Turing Machine ● Space Complexity ○ PSPACE, Savitch's Theorem ● Quantum Computing Model (not algorithms) ● Concurrent Systems (e.g., non-termination, non-determinism, "thread" interference)	Explain	KA	1 1 3 3 2 1 1 1 1 1 1 3 1
AL	Algorithms and Society	● Context-Aware Computing [*corss-reference*: SEP/Social Context] ● Social, Ethical, and Secure Algorithms ● Differential Privacy ● Algorithmic Fairness [*cross-reference* AI] ● Accountability/Transparency	Explain	CS	4

Architecture and Organization (AR)

Preamble

Computing professionals spend considerable time writing efficient code to solve a particular problem in an application domain. With the imminent ending of Moore and Dennard scaling laws, parallelism at the hardware system level has been increasingly utilized to meet performance requirements in almost all systems, including most commodity hardware. This departure from sequential processing demands a more in-depth understanding of the underlying computer architectures. Architecture can no longer be treated as a *black box* where principles from one can be applied to another. Instead, programmers should look inside the black box and use specific components to improve system performance and energy efficiency.

The Architecture and Organization (AR) Knowledge Area aims to develop a deeper understanding of the hardware environments upon which almost all computing is based and the relevant interfaces provided to higher software layers. The target hardware comprises low-end embedded system processors up to high-end enterprise multiprocessors.

The topics in this knowledge area will benefit students by appreciating the fundamental architectural principles of modern computer systems, including the challenge of harnessing parallelism to sustain performance and energy improvements into the future. In addition, students need to comprehend computer architecture to develop programs that can achieve high performance at low energy consumption. This KA will help computer science students depart from the black box approach and become more aware of the underlying computer system and the efficiencies specific architectures can achieve.

Changes since CS 2013: This KA has changed slightly since the CS2013 report. Changes and additions are summarized as follows:

- Topics have been revised, particularly in the Knowledge Units AR/Memory Hierarchy and AR/Performance and Energy Efficiency. This change supports recent advances in-memory caching and energy consumption.
- To address emerging topics in Computer Architecture, the newly created KU AR/Heterogeneous Architectures covers the introductory level: In-Memory processing (PIM), domain-specific architectures (e.g. neural network processors) - and related topics.

- Quantum computing, particularly the development of quantum processor architectures, has been gathering pace recently. The new KU AR/Quantum Architectures offer a "toolbox" covering introductory topics in this important emerging area.
- Knowledge units have been merged to better deal with overlaps:
    - AR/Multiprocessing and Alternative Architectures were merged into newly created AR/Heterogeneous Architectures.

Core Hours

Knowledge Unit	CS Core	KA Core
Digital Logic and Digital Systems		3
Machine-Level Data Representation	1	
Assembly Level Machine Organization	1	2
Memory Hierarchy	6	
Interfacing and Communication	1	
Functional Organization		2
Performance and Energy Efficiency		3
Heterogeneous Architectures		3
Quantum Architectures		3
Total	**9**	**16**

Knowledge Units

AR/Digital Logic and Digital Systems

Topics [KA Core]

- Overview and history of computer architecture
- Combinational vs sequential logic/field programmable gate arrays (FPGAs)
    - Fundamental combinational
    - Sequential logic building block
- Functional hardware and software multi-layer architecture
- Computer-aided design tools that process hardware and architectural representations
- High-level synthesis
    - Register transfer notation
    - Hardware description language (e.g. Verilog/VHDL/Chisel)
- System-on-chip (SoC) design flow
- Physical constraints
    - Gate delays
    - Fan-in and fan-out
    - Energy/power
    - Speed of light

Illustrative learning outcomes [KA Core]

- Comment on the progression of computer technology components from vacuum tubes to VLSI, from mainframe computer architectures to the organization of warehouse-scale computers.
- Comment on parallelism and data dependencies between and within components in a modern heterogeneous computer architecture.
- Explain how the “power wall" makes it challenging to harness parallelism.
- Propose the design of basic building blocks for a computer: arithmetic-logic unit (gate-level), registers (gate-level), central processing unit (register transfer-level), and memory (register transfer-level).
- Evaluate simple building blocks (e.g., arithmetic-logic unit, registers, movement between registers) of a simple computer design.
- Validate the timing diagram behavior of a simple processor, identifying data dependency issues.

AR/Machine-Level Data Representation

Topics [CS Core]

- Bits, bytes, and words
- Numeric data representation and number bases
    - Fixed-point
    - Floating-point

- Signed and twos-complement representations
- Representation of non-numeric data
- Representation of records and arrays

Illustrative learning outcomes [CS Core]

- Comment on the reasons why everything is data in computers, including instructions.
- Follow the diagram and annotate the regions where fixed-length number representations affect accuracy and precision.
- Comment on how negative integers are stored in sign-magnitude and twos-complement representations.
- Articulate with plausible justification how different formats can represent numerical data.
- Explain the bit-level representation of non-numeric data, such as characters, strings, records, and arrays.
- Translate numerical data from one format to another.

AR/Assembly Level Machine Organization

Topics [CS Core]

- von Neumann machine architecture
- Control unit; instruction fetch, decode, and execution
- Introduction to SIMD vs. MIMD and the Flynn taxonomy
- Shared memory multiprocessors/multicore organization

Topics [KA Core]

- Instruction set architecture (ISA) (e.g. x86, ARM and RISC-V)
    - Instruction formats
    - Data manipulation, control, I/O
    - Addressing modes
    - Machine language programming
    - Assembly language programming
- Subroutine call and return mechanisms (xref PL/language translation and execution)
- I/O and interrupts
- Heap, static, stack and code segments

Illustrative learning outcomes [CS Core]

- Contextualize the classical von Neumann functional units in embedded systems, particularly on-chip and off-chip memory.

- Comment on how instruction is executed in a classical von Neumann machine, with extensions for threads, multiprocessor synchronization, and SIMD execution.
- Annotate an example diagram with instruction-level parallelism and hazards to comment on how they are managed in typical processor pipelines.
-

Illustrative learning outcomes [KA Core]

- Comment on how instructions are represented at the machine level and in the context of a symbolic assembler.
- Map an example of high-level language patterns into assembly/machine language notations.
- Comment on different instruction formats, such as addresses per instruction and variable-length vs fixed-length formats.
- Follow a subroutine diagram to comment on how subroutine calls are handled at the assembly level.
- Comment on basic concepts of interrupts and I/O operations.
- Code a simple assembly language program for string processing and manipulation.

AR/Memory Hierarchy

Topics [CS Core]

- Memory hierarchy: the importance of temporal and spatial locality
- Main memory organization and operations
- Persistent memory (e.g. SSD, standard disks)
- Latency, cycle time, bandwidth and interleaving
- Cache memories
    - Address mapping
    - Block size
    - Replacement and store policy
- Multiprocessor cache coherence
- Virtual memory (hardware support, cross-reference OS/Virtual Memory)
- Fault handling and reliability
- Reliability (cross-reference SF/Reliability through Redundancy)
    - Error coding
    - Data compression
    - Data integrity
- Non-von Neumann Architectures
    - In-Memory Processing (PIM)

Illustrative learning outcomes [CS Core]

- Using a memory system diagram, detect the main types of memory technology (e.g., SRAM, DRAM) and their relative cost and performance.
- Measure the effect of memory latency on running time.
- Enumerate the functions of a system with virtual memory management.
- Compute average memory access time under various cache and memory configurations and mixes of instruction and data references.

AR/Interfacing and Communication

Topics [CS Core]

- I/O fundamentals
    - Handshaking and buffering
    - Programmed I/O
    - Interrupt-driven I/O
- Interrupt structures: vectored and prioritized, interrupt acknowledgement
- External storage, physical organization and drives
- Buses fundamentals
    - Bus protocols
    - Arbitration
    - Direct-memory access (DMA)
- Network-on-chip (NoC)

Illustrative learning outcomes [CS Core]

- Follow an interrupt control diagram to comment on how interrupts are used to implement I/O control and data transfers.
- Enumerate various types of buses in a computer system.
- List the advantages of magnetic disks and contrast them with the advantages of solid-state disks.

AR/Functional Organization

Topics [KA Core]

- Implementation of simple datapaths, including instruction pipelining, hazard detection and resolution
- Control unit
    - Hardwired implementation
    - Microprogrammed realization

- Instruction pipelining
- Introduction to instruction-level parallelism (ILP)

Illustrative learning outcomes [KA Core]

- Validate alternative implementation of datapaths in modern computer architectures.
- Propose a set of control signals for adding two integers using hardwired and microprogrammed implementations.
- Comment on instruction-level parallelism using pipelining and significant hazards that may occur.
- Design a complete processor, including datapath and control.
- Compute the average cycles per instruction for a given processor and memory system implementation.

AR/Performance and Energy Efficiency

Topics [KA Core]

- Performance-energy evaluation (introduction): performance, power consumption, memory and communication costs
- Branch prediction, speculative execution, out-of-order execution, Tomasulo's algorithm
- Prefetching
- Enhancements for vector processors and GPUs
- Hardware support for Multithreading
    - Race conditions
    - Lock implementations
    - Point-to-point synchronization
    - Barrier implementation
- Scalability
- Alternative architectures, such as VLIW/EPIC, accelerators and other special-purpose processors
- Dynamic voltage and frequency scaling (DVFS)
- Dark Silicon

Illustrative learning outcomes [KA Core]

- Comment on evaluation metrics for performance and energy efficiency.
- Follow a speculative execution diagram and write about the decisions that can be made.
- Build a GPU performance-watt benchmarking diagram.
- Code a multi-threaded Python program that adds (in parallel) elements of two integer vectors.
- Propose a set of design choices for alternative architectures.
- Comment on key concepts associated with dynamic voltage and frequency scaling.

- Measure improvement of energy savings for an 8-bit integer quantization compared to a 32-bit quantization.

AR/Heterogeneous Architectures

Topics [KA Core]

- SIMD and MIMD architectures (e.g. General-Purpose GPUs, TPUs and NPUs)
- Heterogeneous memory system
    - Shared memory versus distributed memory
    - Volatile vs non-volatile memory
    - Coherence protocols
- Domain-Specific Architectures (DSAs)
    - Machine Learning Accelerator
    - In-networking computing
    - Embedded systems for emerging applications
    - Neuromorphic computing
- Packaging and integration solutions such as 3DIC and Chiplets

Illustrative learning outcomes [KA Core]

- Follow a system diagram with alternative parallel architectures, e.g. SIMD and MIMD, and annotate the key differences.
- Tell what memory-management issues are found in multiprocessors that are not present in uniprocessors, and how these issues might be resolved.
- Validate the differences between memory backplane, processor memory interconnect, and remote memory via networks, their implications for access latency and their impact on program performance.
- Tell how you would determine when to use a domain-specific accelerator instead of just a general-purpose CPU.
- Enumerate key differences in architectural design principles between a vector and scalar-based processing unit.
- List the advantages and disadvantages of PIM architectures.

AR/Quantum Architectures

Topics [KA Core]

- Principles
    - The wave-particle duality principle
    - The uncertainty principle in the double-slit experiment

    - What is a Qubit? Superposition and measurement. Photons as qubits.
    - Systems of two qubits. Entanglement. Bell states. The No-Signaling theorem.
- Axioms of QM: superposition principle, measurement axiom, unitary evolution
- Single qubit gates for the circuit model of quantum computation: X, Z, H.
- Two qubit gates and tensor products. Working with matrices.
- The No-Cloning Theorem. The Quantum Teleportation protocol.
- Algorithms
    - Simple quantum algorithms: Bernstein-Vazirani, Simon's algorithm.
    - Implementing Deutsch-Josza with Mach-Zehnder Interferometers.
    - Quantum factoring (Shor's Algorithm)
    - Quantum search (Grover's Algorithm)
- Implementation aspects
    - The physical implementation of qubits (there are currently nine qubit modalities)
    - Classical control of a Quantum Processing Unit (QPU)
    - Error mitigation and control. NISQ and beyond.
- Emerging Applications
    - Post-quantum encryption
    - The Quantum Internet
    - Adiabatic quantum computation (AQC) and quantum annealing

Illustrative learning outcomes [KA Core]

- Comment on a quantum object produced as a particle, propagates like a wave and is detected as a particle with a probability distribution corresponding to the wave.
- Follow the diagram and comment on the quantum-level nature that is inherently probabilistic.
- Articulate your view on entanglement that can be used to create non-classical correlations, but there is no way to use quantum entanglement to send messages faster than the speed of light.
- Comment on quantum parallelism and the role of constructive vs destructive interference in quantum algorithms given the probabilistic nature of measurement(s).
- Annotate the code snippet provided the role of quantum Fourier sampling (QFT) in Shor's algorithm
- Code Shor's algorithm in a simulator and document your code highlighting the classical components and aspects in Shor's algorithm
- Enumerate the specifics of each qubit modality (e.g., trapped ion, superconducting, silicon spin, photonic, quantum dot, neutral atom, topological, color center, electron-on-helium, etc.)
- Contextualize the differences between AQC and the gate model of quantum computation and which kind of problems each is better suited to solve
- Comment on the statement: a QPU is a heterogeneous multicore architecture like an FPGA or a GPU

Professional Dispositions

- Self-directed: students should increasingly become self-motivated to acquire complementary knowledge from system documentation.
- Proactive: students need to be proactive and independent to navigate and integrate knowledge from different knowledge areas to understand the underlying computer system.
- Inventive: students should look beyond simple solutions to computer architecture design issues and leverage architecture-specific features whenever possible.
- Professional with ethics: students should exercise discretion and behave ethically. Computer systems, particularly embedded sensors, can directly interface with the user's body (e.g. real-time glucose monitoring), so the user's safety and privacy are high-priority.

Math Requirements

Required:

- Discrete Maths: Sets, Relations, Logical Operations, Number Theory
- Linear Algebra: Arithmetic Operations, Matrix operations

Desired:

- Maths/Physics for Quantum Computing: basic probability, trigonometry, simple vector spaces, complex numbers, Euler's formula
- System performance evaluation: probability and factorial experiment design.

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Virtual memory (AR/Memory Hierarchy) with OS/Memory Management.
- Error coding, data compression, and data integrity (AR/Memory Hierarchy) with (SF/Software Reliability)
- Communications networks (AR/Interfacing and Communication) with (NC/Introduction)
- Processor architecture design (AR/Heterogeneous Architectures with Platform-based Development (PBD/Mobile and PBD/Industrial)
- Domain-specific architectures (AR/Heterogeneous Architectures) with Intelligent Systems (IS/Advanced Machine Learning)
- Emerging hardware user interfaces (AR/Heterogeneous Architectures) with Human-Computer Interaction (HCI/Human Factors & Security)
- The underlying parallel computer system (AR/Assembly Level Machine Organization) with Parallel and Distributed Computing (PD/Parallel Architecture)

Crosscutting themes:

- Parallelism at different levels
- Resource allocation, scheduling and isolation
- Performance and Energy Efficiency
- Social and environmental impacts of underlying computer systems

Course Packaging Suggestions

Computer Architecture - Introductory Course to include the following:

- SEP/History [2 hours]
- AR/Machine-Level Data Representation [2 hours]
- AR/Assembly Level Machine Organization [2 hours]
- AR/Memory Hierarchy [10 hours]
- OS/Memory Management [10 hours]
- AR/Interfacing and Communication [4 hours]
- AR/Heterogeneous Architectures [5 hours]
- PD/Programs and Executions [4 hours]
- SEP/Methods for Ethical Analysis [3 hours]

Pre-requisites:

- Discrete Maths: Sets, Relations, Logical Operations, Number Theory

Skill statement:

- A student who completes this course should be able to understand the fundamental architectures of modern computer systems, including the challenge of memory caches and memory management.

Systems Course - Advanced to include the following:

- SEP/History [2 hours]
- SEP/Privacy and Civil Liberties [3 hours]
- SE/Software Design [System Design] [4 hours]
- PD/Algorithms and applications [4 hours]
- AR/Heterogeneous Architectures [4 hours]
- OS/Roles and Purpose of Operating Systems [3 hours]

- AR/Memory Hierarchy [7 hours]
- AR/Performance and Energy Efficiency [5 hours]
- NC/Networked Applications [5 hours]

Pre-requisites:

- Discrete Math: Sets, Relations, Logical Operations, Number Theory

Skill statement:

- A student who completes this course should be able to appreciate the fundamental architectures of modern computer systems, including the challenge of harnessing parallelism to sustain performance and energy improvements into the future.

Competency Specifications

- **Task 1:** Assess the performance implications of cache memories in your application.
- **Competency Statement:** Critically analyze the performance of an application concerning caching issues and produce a report summarizing key results.
- **Competency area:** Systems / Application
- **Competency unit:** Development / Deployment / Evaluation / Improvement
- **Required knowledge areas and knowledge units:**
    - AR/Memory Hierarchy
    - AR/Performance and Energy Efficiency
    - OS/Memory Management
- **Required skill level:** Evaluate
- **Core level:**

- **Task 2:** Evaluate the performance-watt of your machine learning model deployed on an embedded device.
- **Competency Statement:** Evaluate the performance and power consumption of the embedded device running the code deployed with a machine learning model trained in the cloud.
- **Competency area:** Systems / Application
- **Competency unit:** Deployment / Evaluation / Improvement
- **Required knowledge areas and knowledge units:**
    - AR/Performance and Energy Efficiency
    - SPD/Mobile Platforms

- **Required skill level:** Evaluate / Develop
- **Core level:**

- **Task 3:** Develop a version of your CPU-based application to run on a hardware accelerator (GPU, TPU, NPU).
- **Competency Statement:** Apply knowledge from systems design to accelerate an application code and evaluate the code speed-up.
- **Competency area:** Software / Systems
- **Competency unit:** Requirements / Design / Development / Testing / Documentation
- **Required knowledge areas and knowledge units:**
    - AR/Heterogeneous Architectures
    - PD/Parallel Architecture
    - SF/System Design
- **Required skill level:** Evaluate / Develop
- **Core level:**

- **Task 4:** Develop a benchmarking software tool to assess the performance gain in removing I/O bottlenecks in your code.
- **Competency Statement:** Appreciate the importance of taking time to develop or use performance tools to improve the performance of application code.
- **Competency area:** Software / Systems
- **Competency unit:** Development / Evaluation / Maintenance
- **Required knowledge areas and knowledge units:**
    - AR/Interfacing and Communication
    - AR/Performance and Energy Efficiency
    - SF/Resources Allocation and Scheduling
    - SF/System Performance
- **Required skill level:** Evaluate / Explain
- **Core level:**

- **Task 5:** Apply knowledge of operating systems to assess page faults in CPU-GPU memory management and their performance impact on the accelerated application.
- **Competency Statement:**
- **Competency area:** Systems
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Integration / Documentation / Evaluation / Maintenance / Management / Consumer Acceptance / Adaptation to social issues / Improvement

- **Required knowledge areas and knowledge units:**
    - AR/Memory Hierarchy
    - AR/Performance and Energy Efficiency
    - AR/Heterogeneous Architectures
    - OS/Memory Management
- **Required skill level:** Apply (used to be Solve) / Evaluate
- **Core level:**

- **Task 6:** Assess data privacy implications in developing a medical device for continuous patient sensor data collection, and write a white paper on data privacy implications.
- **Competency Statement:** Apply technical knowledge of systems to understand the embedded sensor device and data accuracy vs data minimization and security at the hardware level, and document the data privacy issues.
- **Competency area:** Systems / Application
- **Competency unit:** Requirements / Design / Development / Integration / Documentation / Consumer Acceptance / Adaptation to social issues
- **Required knowledge areas and knowledge units:**
    - AR/Heterogeneous Architectures
    - SEP/Privacy
    - FPL/Embedded Computing and Hardware Interface
- **Required skill level:** Evaluate
- **Core level:**

- **Task 7:** Design software modules for sensor hardware integration.
- **Competency Statement:** Develop high-level software interfaces and low-level hardware interfaces for system integration.
- **Competency area:** Systems
- **Competency unit:** Requirements / Design
- **Required knowledge areas and knowledge units:**
    - AR/Interfacing and Communication
    - AR/Heterogeneous Architectures
    - OS/File Systems API and Implementation
    - SDF/Fundamental Programming Concepts

    - SE/API design principles
    - SDP/Embedded systems
- **Required skill level:** Develop / Evaluate
- **Core level:**

- **Task 8:** Document a system's design choices and proposed system hardware and software architecture.
- **Competency Statement:** Gather application requirements and propose a system architecture solution. Write effective technical documentation.
- **Competency area:** Systems / Application
- **Competency unit:** Requirements / Design / Documentation
- **Required knowledge areas and knowledge units:**
    - AR/Machine-Level Data Representation
    - AR/Performance and Energy Efficiency
    - AR/Heterogeneous Architectures
    - SEP/Professional Communication
    - SE/Product Requirements
- **Required skill level:** Evaluate / Explain
- **Core level:**

- **Task 9:** Work on a multidisciplinary team to effectively develop a sensing-actuator robotics arm for an automated manufacturing cell.
- **Competency Statement:** Cooperate effectively with team members on the design, development and evaluation of a system.
- **Competency area:** Software / Systems / Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Integration / Documentation / Evaluation

- **Required knowledge areas and knowledge units:**
    - AR/Heterogeneous Architectures
    - SPD/Robot Platforms
    - SE/Teamwork
- **Required skill level:** Apply (used to be Solve) / Develop
- **Core level:**

- **Task 10:** Develop a program explicitly exploiting the underlying CPU cores and memory management system for improved performance.
- **Competency Statement:** Understand the computer architecture principles, parallelism and how memory is managed for efficient data sharing.
- **Competency area:** Systems
- **Competency unit:** Requirements / Design / Development / **Required knowledge areas and knowledge units:**
    - AR/Assembly Level Machine Organization
    - AR/Memory Hierarchy
- **Required skill level:** Evaluate / Develop
- **Core level:**

Committee

Chair: Marcelo Pias, Federal University of Rio Grande (FURG), Brazil

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Mohamed Zahran, New York University, NY, USA
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Qiao Xiang, Xiamen University, China
- Adrian German, Indiana University, Bloomington, IN, USA

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
AR	Digital Logic and Digital Systems	● Overview and history of computer architecture ● Combinational vs. sequential logic ○ Fundamental combinational ○ Sequential logic building block ● Functional hardware and software multi-layer architecture	Explain	KA	3
	Digital Logic and Digital Systems	● Computer-aided design tools that process hardware and architectural representations ● High-level synthesis ○ Register transfer notation ○ Hardware description language (e.g. Verilog/VHDL/Chisel) ● System-on-chip (SoC) design flow ● Physical constraints ○ Gate delays ○ Fan-in and fan-out ○ Energy/power [Shared with SPD] ○ Speed of light	Evaluate		
AR	Machine-Level Data Representation	● Bits, bytes, and words ● Numeric data representation and number bases ○ Fixed-point ○ Floating-point ● Signed and twos-complement representations ● Representation of non-numeric data ● Representation of records and arrays	Apply	CS	1
AR	Assembly Level Machine Organization	● von Neumann machine architecture ● Control unit; instruction fetch, decode, and execution ● Introduction to SIMD vs. MIMD and the Flynn taxonomy ● Shared memory multiprocessors/multicore organization [Shared with OS]	Explain	CS	1

AR	Assembly Level Machine Organization	● Instruction set architecture (ISA) (e.g. x86, ARM and RISC-V)
		  ○ Instruction formats
		  ○ Data manipulation, control, I/O
		  ○ Addressing modes
		  ○ Machine language programming
		  ○ Assembly language programming
		● Subroutine call and return mechanisms (xref PL/language translation and execution) [Shared with OS]
		● I/O and interrupts [Shared with OS]
		● Heap, static, stack and code segments [Shared with OS]
			Develop	KA	2

AR	Memory Hierarchy	● Memory hierarchy: the importance of temporal and spatial locality [Shared with OS]
		● Main memory organization and operations
		● Persistent memory (e.g. SSD, standard disks) [Shared with OS]
		● Latency, cycle time, bandwidth and interleaving
		● Virtual memory (hardware support, cross-reference OS/Virtual Memory) [Shared with OS]
		● Fault handling and reliability [Shared with OS]
		● Reliability (cross-reference SF/Reliability through Redundancy)
		  ○ Error coding
		  ○ Data compression
		  ○ Data integrity
		● Non-von Neumann Architectures
		  ○ In-Memory Processing (PIM)
			Explain	CS	6

		● Cache memories [Shared with OS]
		  ○ Address mapping
		  ○ Block size
		  ○ Replacement and store policy
		● Multiprocessor cache coherence
			Evaluate		

AR	Interfacing and Communication	● I/O fundamentals[Shared with OS and SPD]
		  ○ Handshaking and buffering
		  ○ Programmed I/O
		  ○ Interrupt-driven I/O
			Explain	CS	1

		• Interrupt structures: vectored and prioritized, interrupt acknowledgement [Shared with OS]
		• External storage, physical organization and drives [Shared with OS]
		• Buses fundamentals [Shared with OS and SPD]
		  ○ Bus protocols
		  ○ Arbitration
		  ○ Direct-memory access (DMA)
		• Network-on-chip (NoC)			

AR	Functional Organization	• Implementation of simple datapaths, including instruction pipelining, hazard detection and resolution
		• Control unit
		  ○ Hardwired implementation
		  ○ Microprogrammed realization
			Develop	KA	2

		• Instruction pipelining • Introduction to instruction-level parallelism (ILP)	Explain		
AR	Performance and Energy Efficiency	• Performance-energy evaluation (introduction): performance, power consumption, memory and communication costs • Branch prediction, speculative execution, out-of-order execution, Tomasulo's algorithm • Prefetching	Evaluate	KA	2

AR	Performance and Energy Efficiency	• Enhancements for vector processors and GPUs
		• Hardware support for Multithreading
		  ○ Race conditions
		  ○ Lock implementations
		  ○ Point-to-point synchronization
		  ○ Barrier implementation
		• Scalability
		• Alternative architectures, such as VLIW/EPIC, accelerators and other special-purpose processors
		• Dynamic voltage and frequency scaling (DVFS)
			Explain	KA	1

		• Dark Silicon			

AR	Heterogeneous Architectures	• SIMD and MIMD architectures (e.g. General-Purpose GPUs, TPUs and NPUs)
		• Heterogeneous memory system
		  ○ Shared memory versus distributed memory
		  ○ Volatile vs non-volatile memory
		  ○ Coherence protocols
		• Domain-Specific Architectures (DSAs)
		  ○ Machine Learning Accelerator
		  ○ In-networking computing
		  ○ Embedded systems for emerging applications
		  ○ Neuromorphic computing
		• Packaging and integration solutions such as 3DIC and Chiplets
			Explain	KA	3

AR	Quantum Architectures	• Principles
		  • The wave-particle duality principle
		  • The uncertainty principle in the double-slit experiment
		  • What is a Qubit? Superposition and measurement. Photons as qubits.
		  • Systems of two qubits. Entanglement. Bell states. The No-Signaling theorem.
		• Axioms of QM: superposition principle, measurement axiom, unitary evolution
		• Single qubit gates for the circuit model of quantum computation: X, Z, H.
		• Two qubit gates and tensor products. Working with matrices.
		• The No-Cloning Theorem. The Quantum Teleportation protocol.
		• Algorithms
		  • Simple quantum algorithms: Bernstein-Vazirani, Simon's algorithm.
		  • Implementing Deutsch-Josza with Mach-Zehnder Interferometers.
		  • Quantum factoring (Shor's Algorithm)
		  • Quantum search (Grover's Algorithm)
			Explain	KA	3

		● Implementation aspects
		  ● The physical implementation of qubits (there are currently nine qubit modalities)
		  ● Classical control of a Quantum Processing Unit (QPU)
		  ● Error mitigation and control. NISQ and beyond.
		● Emerging Applications
		  ● Post-quantum encryption
		  ● The Quantum Internet
		  ● Adiabatic quantum computation (AQC) and quantum annealing			

Artificial Intelligence (AI)

Preamble

Artificial intelligence (AI) studies problems that are difficult or impractical to solve with traditional algorithmic approaches. These problems are often reminiscent of those considered to require human intelligence, and the resulting AI solution strategies typically generalize over classes of problems. AI techniques are now pervasive in computing, supporting everyday applications such as email, social media, photography, financial markets, and intelligent virtual assistants (e.g., Siri, Alexa). These techniques are also used in the design and analysis of autonomous agents that perceive their environment and interact rationally with it, such as self-driving vehicles and other robots.

Traditionally, AI has included a mix of symbolic and subsymbolic approaches. The solutions it provides rely on a broad set of general and specialized knowledge representation schemes, problem solving mechanisms, and optimization techniques. These approaches deal with perception (e.g., speech recognition, natural language understanding, computer vision), problem solving (e.g., search, planning, optimization), acting (e.g., robotics, task-automation, control), and the architectures needed to support them (e.g., single agents, multi-agent systems). The study of Artificial Intelligence prepares students to determine when an AI approach is appropriate for a given problem, identify appropriate representations and reasoning mechanisms, implement them, and evaluate them with respect to both performance and their broader societal impact.

Over the past decade, the term “artificial intelligence” has become commonplace within businesses, news articles, and everyday conversation, driven largely by a series of high-impact machine learning applications. These advances were made possible by the widespread availability of large datasets, increased computational power, and algorithmic improvements. In particular, there has been a shift from engineered representations to representations learned automatically through optimization over large datasets. The resulting advances have put such terms as “neural networks” and “deep learning” into everyday vernacular. Businesses now advertise AI-based solutions as value-additions to their services, so that “artificial intelligence” is now both a technical term and a marketing keyword. Other disciplines, such as biology, art, architecture, and finance, increasingly use AI techniques to solve problems within their disciplines. For the first time in our history, the broader population has access to sophisticated AI-driven tools, including tools to generate essays or poems from a prompt, photographs or artwork from a description, and fake photographs or videos depicting real people. AI technology is now in widespread use in stock trading, curating our news and social media feeds, automated evaluation of job applicants, detection of medical conditions, and influencing prison sentencing

through recidivism prediction. Consequently, AI technology can have significant societal impacts that must be understood and considered when developing and applying it.

Changes since CS 2013: To reflect this recent growth and societal impact, the knowledge area has been revised from CS 2013 in the following ways:

- The name has changed from “Intelligent Systems” to “Artificial Intelligence,” to reflect the most common terminology used for these topics within the field and its more widespread use outside the field.
- An increased emphasis on neural networks and representation learning reflects the recent advances in the field. Given its key role throughout AI, search is still emphasized but there is a slight reduction on symbolic methods in favor of understanding subsymbolic methods and learned representations. It is important, however, to retain knowledge-based and symbolic approaches within the AI curriculum because these methods offer unique capabilities, are used in practice, ensure a broad education, and because more recent neurosymbolic approaches integrate both learned and symbolic representations.
- There is an increased emphasis on practical applications of AI, including a variety of areas (e.g., medicine, sustainability, social media, etc.). This includes explicit discussion of tools that employ deep generative models (e.g., ChatGPT, DALL-E, Midjourney) and are now in widespread use, covering how they work at a high level, their uses, and their shortcomings/pitfalls.
- The curriculum reflects the importance of understanding and assessing the broader societal impacts and implications of AI methods and applications, including issues in AI ethics, fairness, trust, and explainability.
- The AI knowledge area includes connections to data science through cross-connections with the Data Management knowledge area.
- There are explicit goals to develop basic AI literacy and critical thinking in every computer science student, given the breadth of interconnections between AI and other knowledge areas in practice.

Core Hours

Knowledge Units	CS Core	KA Core
Fundamental Issues	2	1
Search	2 + 3 (AL)	4

Fundamental Knowledge Representation and Reasoning	1 + 1 (MSF)	2
Machine Learning	4	4
Applications and Societal Impact	2	2
Probabilistic Representation and Reasoning		
Planning		
Logical Representation and Reasoning		
Agents		
Natural Language Processing		
Robotics		
Perception and Computer Vision		
Total	**11**	**13**

Knowledge Units

AI/Fundamental Issues

[2 CS Core hours; 1 additional KA hour; Elective Topics]

Topics (CS Core):

- Overview of AI problems, Examples of successful recent AI applications
- Definitions of agents with examples (e.g., reactive, deliberative)
- What is intelligent behavior?
    - The Turing test and its flaws
    - Multimodal input and output
    - Simulation of intelligent behavior
    - Rational versus non-rational reasoning

- Problem characteristics
    - Fully versus partially observable
    - Single versus multi-agent
    - Deterministic versus stochastic
    - Static versus dynamic
    - Discrete versus continuous
- Nature of agents
    - Autonomous, semi-autonomous, mixed-initiative autonomy
    - Reflexive, goal-based, and utility-based
    - Decision making under uncertainty and with incomplete information
    - The importance of perception and environmental interactions
    - Learning-based agents
    - Embodied agents
        - sensors, dynamics, effectors
- AI Applications, growth, and Impact (economic, societal, ethics)

Topics (KA Core):

- Additional depth on problem characteristics with examples
- Additional depth on nature of agents with examples
- Additional depth on AI Applications, growth, and Impact (economic, societal, ethics)

Topics (Elective):

- Philosophical issues.

Learning Outcomes:

1. Describe the Turing test and the “Chinese Room” thought experiment.
2. Differentiate between optimal reasoning/behavior and human-like reasoning/behavior.
3. Determine the characteristics of a specific problem.

AI/Search

[5 CS Core hours, 3 of which overlap with AL Algorithms (Uninformed search); 3 additional KA hours, Elective Topics]

(Cross-reference AL/Basic Analysis, AL/Algorithmic Strategies, AL/Fundamental Data Structures and Algorithms)

Note that the general topics of Branch-and-bound and Dynamic Programming are listed in (AL/Algorithmic Strategies).

Topics (CS Core):

- State space representation of a problem
    - Specifying states, goals, and operators
    - Factoring states into representations (hypothesis spaces)
    - Problem solving by graph search
        - e.g., Graphs as a space, and tree traversals as exploration of that space
        - Dynamic construction of the graph (you're not given it upfront)
- Uninformed graph search for problem solving
    - Breadth-first search
    - Depth-first search
        - With iterative deepening
    - Uniform cost search
- Heuristic graph search for problem solving
    - Heuristic construction and admissibility
    - Hill-climbing
    - Local minima and the search landscape
        - Local vs global solutions
    - Greedy best-first search
    - A* search
- Space and time complexities of graph search algorithms

Topics (KA Core):

- Bidirectional search
- Beam search
- Two-player adversarial games
    - Minimax search
    - Alpha-beta pruning
        - Ply cutoff
- Implementation of A* search

Topics (Elective):

- Understanding the search space
    - Constructing search trees
    - Dynamic search spaces
    - Combinatorial explosion of search space
    - Search space topology (ridges, saddle points, local minima, etc.)
- Local search and constraint satisfaction
- Tabu search
- Variations on A* (IDA*, SMA*, RBFS)
- Two-player adversarial games
    - The horizon effect
    - Opening playbooks / endgame solutions
    - What it means to "solve" a game (e.g., checkers)

- Implementation of minimax search, beam search
- Expectimax search (MDP-solving) and chance nodes
- Stochastic search
    - Simulated annealing
    - Genetic algorithms
    - Monte-Carlo tree search

Learning Outcomes:

1. Design the state space representation for a puzzle (e.g., N-queens or 3-jug problem)
2. Select and implement an appropriate uninformed search algorithm for a problem (e.g., tic-tac-toe), and characterize its time and space complexities.
3. Select and implement an appropriate informed search algorithm for a problem after designing a helpful heuristic function (e.g., a robot navigating a 2D gridworld).
4. Evaluate whether a heuristic for a given problem is admissible/can guarantee an optimal solution.
5. Apply minimax search in a two-player adversarial game (e.g., connect four), using heuristic evaluation at a particular depth to compute the scores to back up. [KA core]
6. Design and implement a genetic algorithm solution to a problem.
7. Design and implement a simulated annealing schedule to avoid local minima in a problem.
8. Design and implement A*/beam search to solve a problem, and compare it against other search algorithms in terms of the solution cost, number of nodes expanded, etc.
9. Apply minimax search with alpha-beta pruning to prune search space in a two-player adversarial game (e.g., connect four).
10. Compare and contrast genetic algorithms with classic search techniques, explaining when it is most appropriate to use a genetic algorithm to learn a model versus other forms of optimization (e.g., gradient descent).
11. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem.

AI/Fundamental Knowledge Representation and Reasoning

[2 CS Core hours; 2 KA Core hours]

Topics (CS Core):

- Types of representations
    - Symbolic, logical
        - Creating a representation from a natural language problem statement
    - Learned subsymbolic representations
    - Graphical models (e.g., naive Bayes, Bayes net)
- Review of probabilistic reasoning, Bayes theorem (cross-reference with DS/Discrete Probability)
- Bayesian reasoning

- Bayesian inference

Topics (KA Core):

- Random variables and probability distributions
  - Axioms of probability
  - Probabilistic inference
  - Bayes' Rule (derivation)
  - Bayesian inference (more complex examples)
- Independence
- Conditional Independence
- Utility and decision making

Learning Outcomes:

1. Given a natural language problem statement, encode it as a symbolic or logical representation.
2. Explain how we can make decisions under uncertainty, using concepts such as Bayes theorem and utility.
3. Make a probabilistic inference in a real-world problem using Bayes' theorem to determine the probability of a hypothesis given evidence.
4. Apply Bayes' rule to determine the probability of a hypothesis given evidence.
5. Compute the probability of outcomes and test whether outcomes are independent.

AI/Machine Learning

[4 CS Core hours; 4 additional KA hours; elective topics]

Topics (CS Core):

- Definition and examples of a broad variety of machine learning tasks
  - Supervised learning
    - Classification
    - Regression
  - Reinforcement learning
  - Unsupervised learning
    - Clustering
- Fundamental ideas:
  - no free lunch
  - undecidability
  - sources of error in machine learning
- Simple statistical-based supervised learning such as Naive Bayes, Decision trees

    - Focus on how they work without going into mathematical or optimization details; enough to understand and use existing implementations correctly
- The overfitting problem and controlling solution complexity (regularization, pruning – intuition only)
    - The bias (underfitting) - variance (overfitting) tradeoff
- Working with Data
    - Data preprocessing
        - Importance and pitfalls of
    - Handling missing values (imputing, flag-as-missing)
        - Implications of imputing vs flag-as-missing
    - Encoding categorical variables, encoding real-valued data
    - Normalization/standardization
    - Emphasis on real data, not textbook examples
- Representations
    - Hypothesis spaces and complexity
    - Simple basis feature expansion, such as squaring univariate features
    - Learned feature representations
- Machine learning evaluation
    - Separation of train, validation, and test sets
    - Performance metrics for classifiers
    - Estimation of test performance on held-out data
    - Tuning the parameters of a machine learning model with a validation set
    - Importance of understanding what your model is actually doing, where its pitfalls/shortcomings are, and the implications of its decisions
- Basic neural networks
    - Fundamentals of understanding how neural networks work and their training process, without details of the calculations
- Ethics for Machine Learning
    - Focus on real data, real scenarios, and case studies.
    - Dataset/algorithmic/evaluation bias

Topics (KA core):

- Formulation of simple machine learning as an optimization problem, such as least squares linear regression or logistic regression
    - Objective function
    - Gradient descent
    - Regularization to avoid overfitting (mathematical formulation)
- Ensembles of models
    - Simple weighted majority combination
- Deep learning
    - Deep feed-forward networks (intuition only, no math)
    - Convolutional neural networks (intuition only, no math)
    - Visualization of learned feature representations from deep nets

- Performance evaluation
    - Other metrics for classification (e.g., error, precision, recall)
    - Performance metrics for regressors
    - Confusion matrix
    - Cross-validation
        - Parameter tuning (grid/random search, via cross-validation)
- Overview of reinforcement learning
- Two or more applications of machine learning algorithms
    - E.g., medicine and health, economics, vision, natural language, robotics, game play
- Ethics for Machine Learning
    - Continued focus on real data, real scenarios, and case studies.
    - Privacy
    - Fairness

Topics (Elective):

- General statistical-based learning, parameter estimation (maximum likelihood)
- Supervised learning
    - Decision trees
    - Nearest-neighbor classification and regression
    - Learning simple neural networks / multi-layer perceptrons
    - Linear regression
    - Logistic regression
    - Support vector machines (SVMs) and kernels
    - Gaussian Processes
- Overfitting
    - The curse of dimensionality
    - Regularization (math computations, L_2 and L_1 regularization)
- Experimental design
    - Data preparation (e.g., standardization, representation, one-hot encoding)
    - Hypothesis space
    - Biases (e.g., algorithmic, search)
    - Partitioning data: stratification, training set, validation set, test set
    - Parameter tuning (grid/random search, via cross-validation)
    - Performance evaluation
        - Cross-validation
        - Metric: error, precision, recall, confusion matrix
        - Receiver operating characteristic (ROC) curve and area under ROC curve
- Bayesian learning (Cross-Reference AI/Reasoning Under Uncertainty)
    - Naive Bayes and its relationship to linear models
    - Bayesian networks
    - Prior/posterior

    - Generative models
- Deep learning
    - Deep feed-forward networks
    - Neural tangent kernel and understanding neural network training
    - Convolutional neural networks
    - Autoencoders
    - Recurrent networks
    - Representations and knowledge transfer
    - Adversarial training and generative adversarial networks
- Representations
    - Manually crafted representations
    - Basis expansion
    - Learned representations (e.g., deep neural networks)
- Unsupervised learning and clustering
    - K-means
    - Gaussian mixture models
    - Expectation maximization (EM)
    - Self-organizing maps
- Semi-supervised learning
- Graphical models (Cross-reference AI/Probabilistic Representation and Reasoning)
- Ensembles
    - Weighted majority
    - Boosting/bagging
    - Random forest
    - Gated ensemble
- Learning theory
    - General overview of learning theory / why learning works
    - VC dimension
    - Generalization bounds
- Reinforcement learning
    - Exploration vs. exploitation trade-off
    - Markov decision processes
    - Value and policy iteration
    - Policy gradient methods
    - Deep reinforcement learning
- Recommender systems
- Hardware for machine learning
    - GPUs / TPUs
- Application of machine learning algorithms to:
    - Medicine and health
    - Economics
    - Education
    - Vision
    - Natural language

    - Robotics
    - Game play
    - Data mining (Cross-reference IM/Data Mining)
- Ethics for Machine Learning
    - Continued focus on real data, real scenarios, and case studies.
    - In depth exploration of dataset/algorithmic/evaluation bias, data privacy, and fairness
    - Trust / explainability

Learning Outcomes:

1. Describe the differences among the three main styles of learning: supervised, reinforcement, and unsupervised.
2. Differentiate the terms of AI, machine learning, and deep learning.
3. Frame an application as a classification problem, including the available input features and output to be predicted (e.g., identifying alphabetic characters from pixel grid input).
4. Apply two or more simple statistical learning algorithms (such as k-nearest-neighbors and logistic regression) to a classification task and measure the classifiers' accuracy.
5. Identify overfitting in the context of a problem and learning curves and describe solutions to overfitting.
6. Explain how machine learning works as an optimization/search process.
7. Implement a statistical learning algorithm and the corresponding optimization process to train the classifier and obtain a prediction on new data.
8. Describe the neural network training process and resulting learned representations
9. Explain proper ML evaluation procedures, including the differences between training and testing performance, and what can go wrong with the evaluation process leading to inaccurate reporting of ML performance.
10. Compare two machine learning algorithms on a dataset, implementing the data preprocessing and evaluation methodology (e.g., metrics and handling of train/test splits) from scratch.
11. Visualize the training progress of a neural network through learning curves in a well-established toolkit (e.g., TensorBoard) and visualize the learned features of the network.
12. Implement simple algorithms for supervised learning, reinforcement learning, and unsupervised learning.
13. Determine which of the three learning styles is appropriate to a particular problem domain.
14. Compare and contrast each of the following techniques, providing examples of when each strategy is superior: decision trees, logistic regression, naive Bayes, neural networks, and belief networks.
15. Evaluate the performance of a simple learning system on a real-world dataset.
16. Characterize the state of the art in learning theory, including its achievements and its shortcomings.
17. Explain the problem of overfitting, along with techniques for detecting and managing the problem.

18. Explain the triple tradeoff among the size of a hypothesis space, the size of the training set, and performance accuracy.

AI/Applications and Societal Impact

[2 CS Core hour; 2 additional KA hours]

(Crosslist with SP)

Note: There is substantial benefit to studying applications and ethics/fairness/trust/explainability in a curriculum alongside the methods and theory that it applies to, rather than covering ethics in a separate, dedicated class session. Whenever possible, study of these topics should be integrated alongside other modules, such as exploring how decision trees could be applied to a specific problem in environmental sustainability such as land use allocation, then assessing the social/environmental/ethical implications of doing so.

For the CS core, cover at least one application and an overview of the societal issues of AI/ML. The KA core should go more in-depth with one or more additional applications, more in-depth on deep generative models, and an analysis and discussion of the social issues.

Topics:

- Applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose one for CS Core, at least one additional for KA core)
    - Formulating and evaluating a specific application as an AI problem
    - Data availability and cleanliness
        - Basic data cleaning and preprocessing
        - Data set bias
    - Algorithmic bias
    - Evaluation bias
- Deployed deep generative models
    - High-level overview of deep image generative models (e.g. as of 2023, DALL-E, Midjourney, Stable Diffusion, etc.), how they work, their uses, and their shortcomings/pitfalls.
    - High-level overview of large language models (e.g. as of 2023, ChatGPT, Bard, etc.), how they work, their uses, and their shortcomings/pitfalls.
- Societal impact of AI
    - Ethics
    - Fairness
    - Trust / explainability

Learning Outcomes:

1. Given a real-world application domain and problem, formulate an AI solution to it, identifying proper data/input, preprocessing, representations, AI techniques, and evaluation metrics/methodology.
2. Analyze the societal impact of one or more specific real-world AI applications, identifying issues regarding ethics, fairness, bias, trust, and explainability.
3. Describe some of the failure modes of current deep generative models for language or images, and how this could affect their use in an application.

AI/Logical Representation and Reasoning ***[Elective]***

Topics:

- Review of propositional and predicate logic (cross-reference DS/Basic Logic)
- Resolution and theorem proving (propositional logic only)
    - Forward chaining, backward chaining
- Knowledge representation issues
    - Description logics
    - Ontology engineering
- Semantic web
- Non-monotonic reasoning (e.g., non-classical logics, default reasoning)
- Argumentation
- Reasoning about action and change (e.g., situation and event calculus)
- Temporal and spatial reasoning
- Logic programming
    - Prolog, Answer Set Programming
- Rule-based Expert Systems
- Semantic networks
- Model-based and Case-based reasoning

Learning Outcomes:

1. Translate a natural language (e.g., English) sentence into a predicate logic statement.
2. Convert a logic statement into clausal form.
3. Apply resolution to a set of logic statements to answer a query.
4. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses.
5. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanism for belief systems.
6. Compare and contrast the basic techniques for representing uncertainty.
7. Compare and contrast the basic techniques for qualitative representation.
8. Apply situation and event calculus to problems of action and change.
9. Explain the distinction between temporal and spatial reasoning, and how they interrelate.

10. Explain the difference between rule-based, case-based and model-based reasoning techniques.
11. Define the concept of a planning system and how it differs from classical search techniques.
12. Describe the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable.
13. Explain the distinction between monotonic and non-monotonic inference.

AI/Probabilistic Representation and Reasoning *[Elective]*

Topics:

- Conditional Independence review
- Knowledge representations
    - Bayesian Networks
        - Exact inference and its complexity
        - Markov blankets and d-separation
        - Randomized sampling (Monte Carlo) methods (e.g. Gibbs sampling)
    - Markov Networks
    - Relational probability models
    - Hidden Markov Models
- Decision Theory
    - Preferences and utility functions
    - Maximizing expected utility

Learning Outcomes:

1. Compute the probability of a hypothesis given the evidence in a Bayesian network.
2. Explain how conditional independence assertions allow for greater efficiency of probabilistic systems.
3. Identify examples of knowledge representations for reasoning under uncertainty.
4. State the complexity of exact inference. Identify methods for approximate inference.
5. Design and implement at least one knowledge representation for reasoning under uncertainty.
6. Describe the complexities of temporal probabilistic reasoning.
7. Design and implement an HMM as one example of a temporal probabilistic system.
8. Describe the relationship between preferences and utility functions.
9. Explain how utility functions and probabilistic reasoning can be combined to make rational decisions.

AI/Planning *[Elective]*

Topics:

- Review of propositional and first-order logic

- Planning operators and state representations
- Total order planning
- Partial-order planning
- Plan graphs and GraphPlan
- Hierarchical planning
- Planning languages and representations
    - PDDL
- Multi-agent planning
- MDP-based planning
- Interconnecting planning, execution, and dynamic replanning
    - Conditional planning
    - Continuous planning
    - Probabilistic planning

Learning Outcomes:

1. Construct the state representation, goal, and operators for a given planning problem.
2. Encode a planning problem in PDDL and use a planner to solve it.
3. Given a set of operators, initial state, and goal state, draw the partial-order planning graph and include ordering constraints to resolve all conflicts
4. Construct the complete planning graph for GraphPlan to solve a given problem.

AI/Agents *[Elective]*

(Cross-reference HCI/Collaboration and Communication)

Topics:

- Agent architectures (e.g., reactive, layered, cognitive)
- Agent theory (including mathematical formalisms)
- Rationality, Game Theory
    - Decision-theoretic agents
    - Markov decision processes (MDP)
- Software agents, personal assistants, and information access
    - Collaborative agents
    - Information-gathering agents
    - Believable agents (synthetic characters, modeling emotions in agents)
- Learning agents
- Multi-agent systems
    - Collaborating agents
    - Agent teams
    - Competitive agents (e.g., auctions, voting)
    - Swarm systems and biologically inspired models

        - Multi-agent learning
- Human-agent interaction
    - Communication methodologies
    - Practical issues
    - Applications
        - Trading agents, supply chain management

Learning Outcomes:

1. Characterize and contrast the standard agent architectures.
2. Describe the applications of agent theory to domains such as software agents, personal assistants, and believable agents.
3. Describe the primary paradigms used by learning agents.
4. Demonstrate using appropriate examples how multi-agent systems support agent interaction.
5. Construct an intelligent agent using a well-established cognitive architecture (ACT-R, SOAR) for solving a specific problem.

AI/Natural Language Processing *[Elective]*

(Cross-reference HCI/New Interactive Technologies)

Topics:

- Deterministic and stochastic grammars
- Parsing algorithms
    - CFGs and chart parsers (e.g. CYK)
    - Probabilistic CFGs and weighted CYK
- Representing meaning / Semantics
    - Logic-based knowledge representations
    - Semantic roles
    - Temporal representations
    - Beliefs, desires, and intentions
- Corpus-based methods
- N-grams and HMMs
- Smoothing and backoff
- Examples of use: POS tagging and morphology
- Information retrieval (Cross-reference IM/Information Storage and Retrieval)
    - Vector space model
        - TF & IDF
    - Precision and recall
- Information extraction
- Language translation

- Text classification, categorization
    - Bag of words model
- Deep learning for NLP (Cross-reference AI/Advanced Machine Learning)
    - RNNs
    - Transformers
    - Multi-modal embeddings (e.g., images + text)
    - Generative language models

Learning Outcomes:

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each.
2. Simulate, apply, or implement classic and stochastic algorithms for parsing natural language.
3. Identify the challenges of representing meaning.
4. List the advantages of using standard corpora. Identify examples of current corpora for a variety of NLP tasks.
5. Identify techniques for information retrieval, language translation, and text classification.
6. Implement a TF/IDF transform, use it to extract features from a corpus, and train an off-the-shelf machine learning algorithm using those features to do text classification.

AI/Robotics *[Elective]*

(Cross-reference HCI)

Topics:

- Overview: problems and progress
    - State-of-the-art robot systems, including their sensors and an overview of their sensor processing
    - Robot control architectures, e.g., deliberative vs. reactive control and Braitenberg vehicles
    - World modeling and world models
    - Inherent uncertainty in sensing and in control
- Sensors and effectors
    - Sensors: LIDAR, sonar, vision, depth, stereoscopic, event cameras, microphones, haptics, etc.
    - Effectors: wheels, arms, grippers, etc.
- Coordinate frames, translation, and rotation (2D and 3D)
- Configuration space and environmental maps
- Interpreting uncertain sensor data
- Localization and mapping
- Navigation and control

- Forward and inverse kinematics
- Motion path planning and trajectory optimization
- Joint control and dynamics
- Vision-based control
- Multiple-robot coordination and collaboration
- Human-robot teaming
    - Shared workspaces
    - Motion/task/goal prediction
    - Collaboration and communication (explicit vs implicit, verbal or symbolic vs non-verbal or visual)
    - Trust

Learning Outcomes:

(Note: Due to the expense of robot hardware, all of these could be done in simulation or with low-cost educational robotic platforms.)

1. List capabilities and limitations of today's state-of-the-art robot systems, including their sensors and the crucial sensor processing that informs those systems.
2. Integrate sensors, actuators, and software into a robot designed to undertake some task.
3. Program a robot to accomplish simple tasks using deliberative, reactive, and/or hybrid control architectures.
4. Implement fundamental motion planning algorithms within a robot configuration space.
5. Characterize the uncertainties associated with common robot sensors and actuators; articulate strategies for mitigating these uncertainties.
6. List the differences among robots' representations of their external environment, including their strengths and shortcomings.
7. Compare and contrast at least three strategies for robot navigation within known and/or unknown environments, including their strengths and shortcomings.
8. Describe at least one approach for coordinating the actions and sensing of several robots to accomplish a single task.
9. Compare and contrast a multi-robot coordination and a human-robot collaboration approach, and attribute their differences to differences between the problem settings.

AI/Perception and Computer Vision *[Elective]*

Topics:

- Computer vision
    - Image acquisition, representation, processing and properties
    - Shape representation, object recognition, and segmentation
    - Motion analysis
    - Generative models
- Audio and speech recognition
- Touch and proprioception

- Other modalities (e.g., olfaction)
- Modularity in recognition
- Approaches to pattern recognition. [cross-reference AI/Machine Learning]
    - Classification algorithms and measures of classification quality
    - Statistical techniques
    - Deep learning techniques

Learning Outcomes:

1. Summarize the importance of image and object recognition in AI and indicate several significant applications of this technology.
2. List at least three image-segmentation approaches, such as thresholding, edge-based and region-based algorithms, along with their defining characteristics, strengths, and weaknesses.
3. Implement 2d object recognition based on contour- and/or region-based shape representations.
4. Distinguish the goals of sound-recognition, speech-recognition, and speaker-recognition and identify how the raw audio signal will be handled differently in each of these cases.
5. Provide at least two examples of a transformation of a data source from one sensory domain to another, e.g., tactile data interpreted as single-band 2d images.
6. Implement a feature-extraction algorithm on real data, e.g., an edge or corner detector for images or vectors of Fourier coefficients describing a short slice of audio signal.
7. Implement an algorithm combining features into higher-level percepts, e.g., a contour or polygon from visual primitives or phoneme hypotheses from an audio signal.
8. Implement a classification algorithm that segments input percepts into output categories and quantitatively evaluates the resulting classification.
9. Evaluate the performance of the underlying feature-extraction, relative to at least one alternative possible approach (whether implemented or not) in its contribution to the classification task (8), above.
10. Describe at least three classification approaches, their prerequisites for applicability, their strengths, and their shortcomings.
11. Implement and evaluate a deep learning solution to problems in computer vision, such as object or scene recognition.

Professional Dispositions

- **Meticulousness:** Attention must be paid to details when implementing AI and machine learning algorithms, requiring students to be meticulous to detail.
- **Persistence:** AI techniques often operate in partially observable environments and optimization processes may have cascading errors from multiple iterations. Getting AI techniques to work predictably takes trial and error, and repeated effort. These call for persistence on the part of the student.

- **Responsible:** Applications of AI can have significant impacts on society, affecting both individuals and large populations. This calls for students to understand the implications of work in AI to society, and to make responsible choices for when and how to apply AI techniques.

Math Requirements

Required:

- Discrete Math:
  - sets, relations, functions
  - predicate and first-order logic, logic-based proofs
- Linear Algebra:
  - Matrix operations, matrix algebra
  - Basis sets
- Probability and Statistics:
  - Basic probability theory, conditional probability, independence
  - Bayes theorem and applications of Bayes theorem
  - Expected value, basic descriptive statistics, distributions
  - Basic summary statistics and significance testing
  - All should be applied to real decision making examples with real data, not "textbook" examples

Desirable:

- Calculus-based probability and statistics
- Other topics in probability and statistics
  - Hypothesis testing, data resampling, experimental design techniques
- Optimization

Shared Concepts and Crosscutting Themes

Shared concepts:

- Search algorithms (AI/Basic Search) with Algorithms and Complexity (AL/Search)
- Data management (for data science) with TBD
- Ethics with TBD
- Robotics with SPD/Industrial Platforms

Crosscutting themes:

- Efficiency
- Ethics
- Modeling

- Programming

Competency Specifications

- **Task 1:** Create a tool to optimize inventory management for salespeople.
- **Competency Statement:** Formulate the problem as search, identify constraints or design admissible heuristics, and implement intelligent search (e.g., A*, constraint satisfaction), as appropriate, to solve the problem.
- **Competency area:** Application
- **Competency unit:** Design / Development / Evaluation
- **Required knowledge areas and knowledge units:**
    - AI/Search
- **Required skill level:** Apply / Develop
- **Core level:**

- **Task 2:** Implement an agent to make strategic decisions in a two-player adversarial game with uncertain actions (e.g., a board game, strategic stock purchasing).
- **Competency Statement:** Use minimax with alpha-beta pruning, and possible chance nodes (expectiminimax), and heuristic move evaluation (at a particular depth) to solve a two-player zero-sum game.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment
- **Required knowledge areas and knowledge units:**
    - AI/Search
    - AI/Fundamental Knowledge Representation and Reasoning
- **Required skill level:** Apply / Develop
- **Core level:**

- **Task 3:** Write a problem that predicts the probability of disease given health data.
- **Competency Statement:** Formulate a problem as Bayesian reasoning and implement probabilistic inference to solve that problem.
- **Competency area:** Application
- **Competency unit:** Development / Testing
- **Required knowledge areas and knowledge units:**
    - AI/Search
    - AI/Logical Representation and Reasoning

    - Math/Probability
- **Required skill level:** Apply / Develop
- **Core level:**

- **Task 4:** Solve a logistics problem.
- **Competency Statement:** Specify the problem as a set of operators and constraints, then solve it using partial order or hierarchical task network planning.
- **Competency area:** Application
- **Competency unit:** Design / Explain
- **Required knowledge areas and knowledge units:**
    - AI/Logical Representation and Reasoning
    - AI/Planning
- **Required skill level:** Explain / Apply
- **Core level:**

- **Task 5:** Analyze tabular data (e.g., customer purchases) to identify trends and predict variables of interest.
- **Competency Statement:** Use machine learning libraries, data preprocessing, training infrastructures, and evaluation methodologies to create a basic supervised learning pipeline
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Evaluation / Deployment
- **Required knowledge areas and knowledge units:**
    - AI/Machine Learning
    - AI/Applications and Societal Impact
- **Required skill level:** Apply / Develop
- **Core level:**

- **Task 6:** Create a model to recognize products in photos.
- **Competency Statement:** Use modern neural network learning techniques on a problem, including fine-tuning a pretrained model, evaluate the quality of the model, and deploy the model.
- **Competency area:** Application

- **Competency unit:** Requirements / Design / Development / Testing / Evaluation / Deployment
- **Required knowledge areas and knowledge units:**
    - AI/Machine Learning
    - AI/Applications and Societal Impact
- **Required skill level:** Apply / Develop
- **Core level:**

- **Task 7:** Implement an intelligent adversary for a video game.
- **Competency Statement:** Understand and apply modern deep reinforcement learning libraries to a visual task.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Evaluation / Deployment
- **Required knowledge areas and knowledge units:**
    - AI/Machine Learning
- **Required skill level:** Apply / Develop
- **Core level:**

- **Task 8:** Develop a tool for identifying the sentiment of social media posts.
- **Competency Statement:** Extract features from a text corpus, and train an off-the-shelf machine learning algorithm using those features to do text classification.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Evaluation / Deployment
- **Required knowledge areas and knowledge units:**
    - AI/Machine Learning
    - AI/Natural Language Processing

    - AI/Applications and Societal Impact
- **Required skill level:** Apply / Develop
- **Core level:**

- **Task 9:** Critique a deployed machine learning model in terms of potential bias and correct the issues.
- **Competency Statement:** Understand, recognize, and evaluate issues of data set bias in AI, the types of bias, and algorithmic strategies for mitigation.
- **Competency area:** Application
- **Competency unit:** Communication
- **Required knowledge areas and knowledge units:**
    - AI/Machine Learning
    - AI/Applications and Societal Impact
- **Required skill level:** Understand and explain
- **Core level:**

- **Task 10:** Automate first-level customer service using a large language model.
- **Competency Statement:** Understand and use a deep generative model, understanding their potential failure modes and how this can and should affect their use.
- **Competency area:** Application
- **Competency unit:** Communication
- **Required knowledge areas and knowledge units:**
    - AI/Applications and Societal Impact
- **Required skill level:** Understand and explain
- **Core level:**

Course Packaging Suggestions

Artificial Intelligence to include the following:

- AI/Fundamental Issues: 4 hrs
- AI/Search: 9 hrs
- AI/Fundamental knowledge representation and reasoning: 4 hrs

- AI/Probabilistic knowledge representation and reasoning: 5 hrs
- AI/Machine Learning: 12 hrs
- AI/Applications and societal impact: 4 hrs (should be integrated throughout the course)

Prerequisites:

- CS2
- Discrete math
- Probability

Skill statement: A student who completes this course should understand the basic areas of AI and be able to understand, develop, and apply techniques in each. They should be able to solve problems using search techniques, basic Bayesian reasoning, and simple machine learning methods. They should understand the various applications of AI and associated ethical and societal implications.

Machine Learning to include the following:

- AI/Machine Learning: 32 hrs
- AI/Fundamental Knowledge Representation and Reasoning: 4 hrs
- AI/Natural Language Processing: 4 hrs (selected topics, e.g., TF-IDF, bag of words, and text classification)
- AI/Applications and societal impact: 4 hrs (should be integrated throughout the course)

Prerequisites:

- CS2
- Discrete math
- Probability
- Statistics
- Linear algebra (optional)

Skill statement: A student who completes this course should be able to understand, develop, and apply mechanisms for supervised learning and reinforcement learning. They should be able to select the proper machine learning algorithm for a problem, preprocess the data appropriately, apply proper evaluation techniques, and explain how to interpret the resulting models, including the model's shortcomings. They should be able to identify and compensate for biased data sets and other sources of error, and be able to explain ethical and societal implications of their application of machine learning to practical problems.

Robotics to include the following:

- AI/Robotics: 25 hrs
- SPD/Robot Platforms: 4 hrs (focusing on hardware, constraints/considerations, and software architectures; some other topics in SPD/Robot Platforms overlap with AI/Robotics)
- AI/Search: 4 hrs (selected topics well-integrated with robotics, e.g., A* and path search)

- AI/Machine Learning: 6 hrs (selected topics well-integrated with robotics, e.g., neural networks for object recognition)
- AI/Applications and societal impact: 3 hrs (should be integrated throughout the course; robotics is already a huge application, so this really should focus on societal impact and specific robotic applications)

Prerequisites:

- CS2
- Linear algebra

Skill statement: A student who completes this course should be able to understand and use robotic techniques to perceive the world using sensors, localize the robot based on features and a map, and plan paths and navigate in the world in simple robot applications. They should understand and be able to apply simple computer vision, motion planning, and forward and inverse kinematics techniques.

Committee

Chair: Eric Eaton, University of Pennsylvania, Philadelphia, USA

Members:

- Zachary Dodds, Harvey Mudd College
- Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, New York, NY, USA
- Laura Hiatt, US Naval Research Lab
- Amruth N. Kumar, Ramapo College of New Jersey, Mahwah, USA
- Peter Norvig, Google
- Meinolf Sellmann, GE Research
- Reid Simmons, Carnegie Mellon University

Contributors:

- Claudia Schulz, Thomson Reuters

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours

AI	Fundamental Issues	• Overview of AI problems, Examples of successful recent AI applications
		• Definitions of agents with examples (e.g., reactive, deliberative)
		• What is intelligent behavior?
		  • The Turing test and its flaws
		  • Multimodal input and output
		  • Simulation of intelligent behavior
		  • Rational versus non-rational reasoning
			Explain	CS	2

		• Problem characteristics
		  • Fully versus partially observable
		  • Single versus multi-agent
		  • Deterministic versus stochastic
		  • Static versus dynamic
		  • Discrete versus continuous
		• Nature of agents
		  • Autonomous, semi-autonomous, mixed-initiative autonomy
		  • Reflexive, goal-based, and utility-based
		  • Decision making under uncertainty and with incomplete information
		  • The importance of perception and environmental interactions
		  • Learning-based agents
		  • Embodied agents
		  ▪ sensors, dynamics, effectors
			Evaluate		

		• AI Applications, growth, and Impact (economic, societal, ethics)	Explain		
AI	Fundamental Issues	• Additional depth on problem characteristics with examples • Additional depth on nature of agents with examples • Additional depth on AI Applications, growth, and Impact (economic, societal, ethics)	Evaluate	KA	1
AI	Search	• State space representation of a problem ○ Specifying states, goals, and operators	Explain		

AL *	Fundamental Data Structures and Algorithms *	○ Factoring states into representations (hypothesis spaces) ○ Problem solving by graph search ■ e.g., Graphs as a space, and tree traversals as exploration of that space ■ Dynamic construction of the graph (you're not given it upfront)		CS	5
		● Uninformed graph search for problem solving ○ Breadth-first search ○ Depth-first search ■ With iterative deepening ○ Uniform cost search	Develop/Apply		
		● Heuristic graph search for problem solving ○ Heuristic construction and admissibility ○ Hill-climbing ○ Local minima and the search landscape ■ Local vs global solutions ○ Greedy best-first search ○ A* search	Develop/Apply		
		● Space and time complexities of graph search algorithms	Evaluate		
AI	Search	● Bidirectional search ● Beam search ● Two-player adversarial games ○ Minimax search ○ Alpha-beta pruning ■ Ply cutoff ● Implementation of A* search	Develop/Apply	KA	3
AI	Fundamental Knowledge Represe	● Types of representations ○ Symbolic, logical ▪ Creating a representation from a natural language problem statement	Explain	CS	2

	ntation and Reasoning	○ Learned subsymbolic representations ○ Graphical models (e.g., naive Bayes, Bayes net) ● Review of probabilistic reasoning, Bayes theorem (cross-reference with DS/Discrete Probability)			
		● Bayesian reasoning ○ Bayesian inference	Apply		
AI	Fundamental Knowledge Representation and Reasoning	● Random variables and probability distributions ● Axioms of probability ● Probabilistic inference ● Bayes' Rule (derivation) ● Bayesian inference (more complex examples) ● Independence ● Conditional Independence ● Utility and decision making	Apply	KA	2
AI	Machine Learning	● Definition and examples of a broad variety of machine learning tasks ○ Supervised learning ▪ Classification ▪ Regression ○ Reinforcement learning ○ Unsupervised learning ▪ Clustering ● Fundamental ideas: ○ no free lunch ○ undecidability ○ sources of error in machine learning		CS	4

		• Simple statistical-based supervised learning such as Naive Bayes, Decision trees
		  ○ Focus on how they work without going into mathematical or optimization details; enough to understand and use existing implementations correctly
		• The overfitting problem and controlling solution complexity (regularization, pruning – intuition only)
		  ○ The bias (underfitting) - variance (overfitting) tradeoff
		• Working with Data
		  ○ Data preprocessing
		  ▪ Importance and pitfalls of
		  ○ Handling missing values (imputing, flag-as-missing)
		  ▪ Implications of imputing vs flag-as-missing
		  ○ Encoding categorical variables, encoding real-valued data
		  ○ Normalization/standardization
		  ○ Emphasis on real data, not textbook examples
		• Representations
		  ○ Hypothesis spaces and complexity
		  ○ Simple basis feature expansion, such as squaring univariate features
		  ○ Learned feature representations
		• Machine learning evaluation
		  ○ Separation of train, validation, and test sets
		  ○ Performance metrics for classifiers
		  ○ Estimation of test performance on held-out data
		  ○ Tuning the parameters of a machine learning model with a validation set
		  ○ Importance of understanding what your model is actually doing, where its pitfalls/shortcomings are, and the implications of its decisions
		• Basic neural networks
			Apply/
			Develop/
			Evaluate		

		○ Fundamentals of understanding how neural networks work and their training process, without details of the calculations			
		● Ethics for Machine Learning ○ Focus on real data, real scenarios, and case studies. ○ Dataset/algorithmic/evaluation bias	Explain / Evaluate		
AI	Machine Learning	● Formulation of simple machine learning as an optimization problem, such as least squares linear regression or logistic regression ○ Objective function ○ Gradient descent ○ Regularization to avoid overfitting (mathematical formulation) ● Ensembles of models ○ Simple weighted majority combination ● Deep learning ● Deep feed-forward networks (intuition only, no math) ● Convolutional neural networks (intuition only, no math) ● Visualization of learned feature representations from deep nets ● Performance evaluation ● Other metrics for classification (e.g., error, precision, recall) ● Performance metrics for regressors ● Confusion matrix ● Cross-validation	Apply/ Develop/ Evaluate	KA	4

		▪ Parameter tuning (grid/random search, via cross-validation) ● Overview of reinforcement learning ● Two or more applications of machine learning algorithms ● E.g., medicine and health, economics, vision, natural language, robotics, game play			
		● Ethics for Machine Learning ● Continued focus on real data, real scenarios, and case studies. ● Privacy ● Fairness	Explain/ Evaluate		
AI	Applications and Societal Impact	*Note: For the CS core, cover at least one application and an overview of the societal issues of AI/ML.* ● Applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose one for CS Core, at least one additional for KA core) o Formulating and evaluating a specific application as an AI problem o Data availability and cleanliness ▪ Basic data cleaning and preprocessing ▪ Data set bias o Algorithmic bias o Evaluation bias ● Deployed deep generative models o High-level overview of deep image generative models (e.g. as of 2023,	Explain/ Evaluate	CS	2

		DALL-E, Midjourney, Stable Diffusion, etc.), how they work, their uses, and their shortcomings/pitfalls. ○ High-level overview of large language models (e.g. as of 2023, ChatGPT, Bard, etc.), how they work, their uses, and their shortcomings/pitfalls. ● Societal impact of AI ○ Ethics ○ Fairness ○ Trust / explainability			
AI	Applications and Societal Impact	*Note: The KA core should go more in-depth with one or more additional applications, more in-depth on deep generative models, and an analysis and discussion of the social issues.* ● Applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose one for CS Core, at least one additional for KA core) ○ Formulating and evaluating a specific application as an AI problem ○ Data availability and cleanliness ▪ Basic data cleaning and preprocessing ▪ Data set bias ○ Algorithmic bias ○ Evaluation bias ● Deployed deep generative models ○ High-level overview of deep image generative models (e.g. as of 2023, DALL-E, Midjourney, Stable Diffusion, etc.), how they work, their uses, and their shortcomings/pitfalls. ○ High-level overview of large language models (e.g. as of 2023, ChatGPT, Bard, etc.), how they work, their uses, and their shortcomings/pitfalls.	Explain/ Evaluate	KA	2

		• Societal impact of AI
		  ○ Ethics
		  ○ Fairness
		  ○ Trust / explainability			

Data Management (DM)

Preamble

Each area of computer science can be described as "The study of algorithms and data structures to ..." In this case the blank is filled in with "deal with persistent data sets; frequently too large to fit in primary memory."

Since the mid-1970's this has meant an almost exclusive study of relational database systems. Depending on institutional context, students have studied, in varying proportions:

- Data modeling and database design: e.g. E-R Data model, relational model, normalization theory
- Query construction: e.g. relational algebra, SQL
- Query processing: e.g. indices (B+tree, hash), algorithms (e.g. external sorting, select, project, join), query optimization (transformations, index selection)
- DBMS internals: e.g. concurrency/locking, transaction management, buffer management

Today's graduates are expected to possess DBMS user (as opposed to developer) skills. These primarily include data modeling and query construction; ability to take an unorganized collection of data, organize it using a DBMS, and access/update the collection via queries.

Additionally, students need to study:

- The role data plays in an organization. This includes:
  - The Data Life Cycle: Creation-Processing-Review/Reporting-Retention/Retrieval-Destruction.
  - The social/legal aspects of data collections: e.g. scale, data privacy, database privacy (compliance) by design, de-identification, ownership, reliability, database security, and intended and unintended applications.
- Emerging and advanced technologies that are augmenting/replacing traditional relational systems, particularly those used to support (big) data analytics: NoSQL (e.g. JSON, XML, key-value store databases), cloud databases, MapReduce, dataframes).

We recognize the existing and emerging roles for those involved with data management, which include:

- Product feature engineers: those who use both SQL and NoSQL operational databases.
- Analytical Engineers/Data Engineers: those who write analytical SQL, Python, and Scala code to build data assets for business groups.
- Business Analysts: those who build/manage data most frequently with Excel spreadsheets.
- Data Infrastructure Engineers: those who implement a data management system (e.g. OLTP).
- “Everyone:” those who produce or consume data need to understand the associated social, ethical, and professional issues.

One role that transcends all of the above categories is that of data custodian. Previously, data was seen as a resource to be managed (Information Systems Management) just like other enterprise resources. Today, data is seen in a larger context. Data about customers can now be seen as belonging to (or in

some national contexts, as owned by) those customers. There is now an accepted understanding that the ethical storage and use of institutional data is part of being a responsible data custodian.

Furthermore, we acknowledge the tension between a curricular focus on professional preparation versus the study of a knowledge area as a scientific endeavor. This is particularly true with Data Management. For example, proving (or at least knowing) the completeness of Armstrong's Axioms is fundamental in functional dependency theory. However, the vast majority of computer science graduates will never utilize this concept during their professional careers. The same can be said for many other topics in the Data Management canon. Conversely, if our graduates can only normalize data into Boyce-Codd normal form (using an automated tool) and write SQL queries, without understanding the role that indices play in efficient query execution, we have done a disservice.

To this end, the number of CS Core hours is relatively small relative to the KA Core hours. Hopefully, this will allow institutions with differing contexts to customize their curricula appropriately. For some, the efficient storage and access of data is primary and independent of how the data is ultimately used - institutional context with a focus on OLTP implementation. For others, what is "under the hood" is less important than the programmatic access to already designed databases - institutional context with a focus on product feature engineers/data scientists.

Regardless of how an institution manages this tension we wish to give voice to one of the ironies of computer science curricula. Students typically spend the majority of their educational career reading (and writing) data from a file or interactively, while outside of the academy the lion's share of data, by a huge margin, comes from databases accessed programmatically. Perhaps in the not too distant future students will learn programmatic database access early on and then continue this practice as they progress through their curriculum.

Finally, we understand that while Data Management is orthogonal to Cybersecurity and SEP (Society, Ethics, and Professionalism), it is also ground zero for these (and other) knowledge areas. When designing persistent data stores, the question of what should be stored must be examined from both a legal and ethical perspective. Are there privacy concerns? And finally, how well protected is the data?

Changes since CS 2013:

Core Hours

Knowledge Unit	CS Core Hours	KA Core Hours
The Role of Data	2	
Core Database Systems Concepts	2	.5
Data Modeling	2	3

Relational Databases	1	3
Query Construction	2	4
Query Processing		4
DBMS Internals		4
NoSQL Systems		2
Data Security & Privacy		
Data Analytics		
Distributed Databases/Cloud Computing		
Semi-structured and Unstructured Databases		
Total	**9**	**25**

A knowledge unit is labeled CS Core (red) or KA Core (blue) which encapsulates all the topics within the unit. Finally, some knowledge units/topics are unlabeled and therefore considered "elective," to indicate that while relevant, and an interesting component of this knowledge area, it is not considered a required knowledge unit

Knowledge Units

The Role of Data

Topics:

[CS Core - 2 hours]

- The Data Life Cycle: Creation-Processing-Review/Reporting-Retention/Retrieval-Destruction.
- The social/legal aspects of data collections:
  - scale
  - data privacy
  - database privacy (compliance) by design
  - anonymity
  - ownership

- reliability
- intended and unintended applications

Illustrative Learning Outcomes:

CS Core:

TBD.

Core Database System Concepts

Topics:

[CS Core - 2 hours]

- Purpose and advantages of database systems
- Components of database systems
- Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods)
- Database architecture, data independence, and data abstraction
- Use of a declarative query language
- Transaction mgmt
- Normalization
- Approaches for managing large volumes of data (e.g., noSQL database systems, use of MapReduce).
- How to support CRUD-only applications
- Distributed databases/cloud-based systems
- Structured, semi-structured, and unstructured databases

[KA Core - .5 hours]

- Systems supporting structured and/or stream content

Illustrative Learning Outcomes:

[CS Core]

1. Identify at least four advantages that using a database system provides.
2. Enumerate the components of a (relational) database system.
3. Follow a query as it is processed by the components of a (relational) database system.
4. Defend the value of data independence.
5. Compose a simple select-project-join query in SQL.
6. Enumerate the four properties of a correct transaction manager.
7. Articulate the advantages for eliminating duplicate repeated data.
8. Outline how MapReduce uses parallelism to process data efficiently.
9. Evaluate the differences between structured and semi/unstructured databases.

Data Modeling

Topics:

[CS Core - 2 hours]

- Data modeling
- Relational data model

[KA Core - 3 hours]

- Conceptual models (e.g., entity-relationship, UML diagrams)
- Semi-structured data model (expressed using DTD, XML, or JSON Schema, for example)

[Elective]

- Spreadsheet models
- Object-oriented models (cross-reference PL/Object-Oriented Programming)
    - GraphQL
- What is new in SQL:202x
- Specialized Data Modeling topics
    - Time series data (aggregation, and join)
    - Graph data (link traversal)
    - Materialized Views and Special data structures like (Hyperloglog, bitmap, …)
        - Typically querying “Raw time series data” directly is very slow for things like “avg daily price”, “daily unique count”, “daily membership”
    - Geo-Spatial data

Illustrative Learning Outcomes:

[CS Core]

1. Articulate the components of the relational data model
2. Model 1:1, 1:n, and n:m relationships using the relational data model

[KA Core]

1. Articulate the components of the E-R (or some other non-relational) data model.
2. Model a given environment using a conceptual data model.
3. Model a given environment using the document-based or key-value store-based data model.

Relational Databases

Topics:

[CS Core - 1 hours]

- Entity and referential integrity
  - Candidate key, superkeys
- Relational database design

[KA Core - 3 hours]

- Mapping conceptual schema to a relational schema
- Physical database design: file and storage structures
- Introduction to Functional dependency Theory
- Normalization Theory
  - Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition
  - Normal forms (BCNF)
  - Denormalization (for efficiency)

[Elective]

- Functional dependency Theory
  - Closure of a set of attributes
  - Canonical Cover
- Normalization Theory
  - Multi-valued dependency (4NF)
  - Join dependency (PJNF, 5NF)
  - Representation theory

Illustrative Learning Outcomes:

[CS Core]

1. Articulate the defining characteristics behind the relational data model.
2. Comment on the difference between a foreign key and a superkey.
3. Enumerate the different types of integrity constraints.

[KA Core]

1. Compose a relational schema from a conceptual schema which contains 1:1, 1:n, and n:m relationships.
2. Map appropriate file structure to relations and indices.
3. Articulate how functional dependency theory generalizes the notion of key.
4. Defend a given decomposition as lossless and or dependency preserving.
5. Detect which normal form a given decomposition yields.

6. Comment on reasons for denormalizing a relation.

Query Construction

Topics:

[CS Core - 2 hours]

- SQL Query Formation

[KA Core - 4 hours]

- Relational Algebra
- SQL
    - Data definition including integrity and other constraints specification
    - Update sublanguage

[Elective]

- Relational Calculus
- QBE and 4th-generation environments
- Different ways to invoke non-procedural queries in conventional languages (overlap with PL)
- Introduction to other major query languages (e.g., XPATH, SPARQL)
- Stored procedures

Illustrative Learning Outcomes:

[CS Core]

1. Compose SQL queries that incorporate select, project, join, union, intersection, set difference, and set division.
2. Determine when a nested SQL query is correlated or not.
3. Iterate over data retrieved programmatically from a database via an SQL query.

[KA Core]

1. Define, in SQL, a relation schema, including all integrity constraints and delete/update triggers.
2. Compose an SQL query to update a tuple in a relation.

Query Processing

Topics:

[KA Core - 4 hours]

- Index structures
  - B+ trees
  - Hash indices: static and dynamic
  - Index creation in SQL
- Algorithms for query operators
  - External Sorting
  - Selection
  - Projection; with and without duplicate elimination
  - Natural Joins: Nested loop, Sort-merge, Hash join
  - Analysis of algorithm efficiency
- Query transformations
- Query optimization
  - Access paths
  - Query plan construction
  - Selectivity estimation
  - Index-only plans
- Database tuning: Index selection
  - Impact of indices on query performance

Illustrative Learning Outcomes:

[KA Core]

1. Articulate the purpose and organization of both B+ tree and hash index structures.
2. Compose an SQL query to create an index (any kind).
3. Articulate the steps for the various query operator algorithms: external sorting, projection with duplicate elimination, sort-merge join, hash-join, block nested-loop join.
4. Derive the run-time (in I/O requests) for each of the above algorithms
5. Transform a query in relational algebra to its equivalent appropriate for a left-deep, pipelined execution.
6. Compute selectivity estimates for a given selection and/or join operation.
7. Articulate how to modify an index structure to facilitate an index-only operation for a given relation.
8. For a given scenario decide on which indices to support for the efficient execution of a set of queries.

DBMS Internals

Topics:

[KA Core - 4 hours]

- DB Buffer Management
- Transaction Processing
    - Isolation Levels
    - ACID
    - Serializability
- Concurrency Control:
    - 2-Phase Locking
    - Deadlocks handling strategies
- Recovery Manager
    - Relation with Buffer Manager

[Elective]

- Concurrency Control:
    - Optimistic CC
    - Timestamp CC
- Recovery Manager
    - Write-Ahead logging
    - ARIES recovery system (Analysis, REDO, UNDO)

Illustrative Learning Outcomes:

[KA Core]

1. Articulate how a DBMS manages its Buffer Pool
2. Articulate the four properties for a correct transaction manager
3. Outline the principle of serializability

NoSQL Systems

Topics:

[KA Core - 2 hours]

- Why NoSQL? (e.g. Impedance mismatch between Application [CRUD] and RDBMS)
- Key-Value and Document data model

[Elective]

- Storage system (e.g. Key-Value system)
- Distribution Models (Sharding and Replication)
- Consistency Models (Update and Read, Quorum consistency, CAP theorem)
- Processing model (e.g. Map-Reduce, multi-stage map-reduce, incremental map-reduce)
- Case Studies: Cloud storage system (e.g. S3); Graph databases ; “When not to use NoSQL”

Illustrative Learning Outcomes:

[KA Core]

1. Articulate a use case for the use of NoSQL over RDBMS.
2. Articulate the defining characteristics behind Key-Value and Document-based data models.

Data Security & Privacy

Topics:

Tbd

Illustrative Learning Outcomes:

tbd

Data Analytics

Topics:

tbd

Illustrative Learning Outcomes:

tbd

Distributed Databases/Cloud Computing

Topics:

[Elective]

- Distributed DBMS
  - Distributed data storage
  - Distributed query processing
  - Distributed transaction model
  - Homogeneous and heterogeneous solutions
  - Client-server distributed databases (cross-reference SF/Computational Paradigms)
- Parallel DBMS
  - Parallel DBMS architectures: shared memory, shared disk, shared nothing;

- Speedup and scale-up, e.g., use of the MapReduce processing model (cross-reference CN/Processing, PD/Parallel Decomposition)
- Data replication and weak consistency models

Semi-structured and Unstructured Databases

Topics:

[Elective]

- Vectorized unstructured data (text, video, audio, …) and vector storage
  - TF-IDF Vectorizer with ngram
  - Word2Vec Word2vec - Wikipedia
  - Array database or array data type handling
- semi-structured (e.g. JSON)
  - Storage
    - Encoding and compression of nested data types
  - Indexing
    - Btree, skip index, Bloom filter
    - Inverted index and bitmap compression
    - Space filling curve indexing for semi-structured geo-data
  - Query processing for OLTP and OLAP use cases
    - Insert, Select, update/delete trade offs
    - Case studies on Postgres/JSON, MongoDB and Snowflake/JSON

Professional Dispositions

- **Professional:** This is potentially the most important disposition for Data Management. Though the notion of "data ownership" varies across national contexts, the importance of how an enterprise manages that data does not.
- **Responsible**: In conjunction with the professional management of (personal) data, it is equally important that data be managed responsibly. Protection from unauthorized access as well as prevention of irresponsible, though legal, use of data is paramount. Furthermore, data custodians need to protect data not only from outside attack, but from crashes and other foreseeable dangers.
- **Collaborative**: Data managers and data users must behave in a collaborative fashion to ensure that the correct data is accessed, and is used only in an appropriate manner.
- **Responsive**: The data that gets stored and is accessed is always in response to an institutional need/request.

Math Requirements

Required:

- Set theory (union, intersection, difference, cross-product)

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Society, Ethics, and Professionalism
- Programming Languages
- Systems Fundamentals
- Algorithms and Complexity
- Parallel and Distributed Computing
- Specialized Platform Development
- Security
- Data Mining: See the ACM Data Science curriculum

Course Packaging Suggestions

Competency Specifications

- **Task 1:** Access data from a database for some purpose.
- **Competency Statement:** Access the database programmatically and iterate over the resulting relation performing some computation.
- **Competency area:** Software, Systems
- **Competency unit:** Development, testing
- **Required knowledge areas and knowledge units:**
    - DM / Query Construction
    - SEP / Social Context
    - SEP / Equity, Diversity and Inclusion
    - SEP / Methods for Ethical Analysis

    - SEP / Professional Ethics
- **Required skill level:** Develop
- **Core level:**

- **Task 2:** Produce a white paper assessing the social and ethical implications for collecting and storing the data from a new (or existing) application. (risk)
- **Competency Statement:** Identify the stakeholders and evaluate the potential long-term consequences for the collection and retention of data objects. Consider both potential harm from unintended data use and from data breaches.
- **Competency area:** Systems, Theory
- **Competency unit:** Evaluation, Management, Adaptation to social issues.
- **Required knowledge areas and knowledge units:**
    - SEP / Social Context
    - SEP / Methods for Ethical Analysis
    - SEP / Privacy and Civil Liberties
    - SEP / Professional Ethics
    - SEP / Security Policies, Laws and Computer Crimes
    - SEP / Equity, Diversity and Inclusion
    - DM / The Role of Data
    - SEC / Foundational Security
- **Required skill level:** Evaluate/Explain
- **Core level:**

- **Task 3:** Determine how to store a new application's data.

- **Competency Statement:** Choose and defend the appropriate data model (relational, key-value store, etc) for a given application.
- **Competency area:** Application, Theory
- **Competency unit:** Evaluation, Design
- **Required knowledge areas and knowledge units:**
    - DM / The Role of Data
    - DM / Core Database Systems Concepts
    - DM / Data Modeling
    - DM / Relational Databases
    - DM / NoSQL Systems
    - SEP / Security Policies, Laws and Computer Crimes
- **Required skill level:** Evaluate/Explain
- **Core level:**

- **Task 4:** Improve a database application's performance (speed)
- **Competency Statement:** Remove and/or create index structures to speed up the execution of frequent/important queries which are part of the application.
- **Competency area:** Software, Application
- **Competency unit:** Design, Improvement
- **Required knowledge areas and knowledge units:**
    - DM / Query Processing
    - DM / DBMS Internals
    - PD / Parallel Algorithms, Analysis, and Programming
- **Required skill level:** Evaluate, Develop
- **Core level:**

- **Task 5:** Secure data from unauthorized access
- **Competency Statement:** Create database views to ensure data access is appropriately limited.
- **Competency area:** Management

- **Competency unit:** Maintenance, Management
- **Required knowledge areas and knowledge units:**
    - DM / The Role of Data
    - DM / Relational Databases
    - DM / Query Processing
    - SEP / Security Policies, Laws and Computer Crimes
    - SEP / Professional Ethics
    - SEP / Privacy and Civil Liberties
    - SEC / Foundational Security
- **Required skill level:** Develop
- **Core level:**

- **Task 6:** Get back online after a disruption (e.g. power outage)
- **Competency Statement:** Restore the database to its most recent "safe" state using a recovery manager
- **Competency area:** Systems
- **Competency unit:** Maintenance, Management
- **Required knowledge areas and knowledge units:**
    - DM / DBMS Internals
- **Required skill level:** Learn how to use the appropriate restoration tool / Apply
- **Core level:**

- **Task 7:** Design the data storage for a new application
- **Competency Statement:** Model the application environment using an appropriate data model. If appropriate, convert to the relational model and normalize.
- **Competency area:** Application
- **Competency unit:** Design, Improvement
- **Required knowledge areas and knowledge units:**
    - DM / The Role of Data
    - DM / Data Modeling
    - SEP / Social Context

    - SEP / Methods for Ethical Analysis
    - SEP / Privacy and Civil Liberties
    - SEP / Professional Ethics
    - SEP / Security Policies, Laws and Computer Crimes
    - SEP / Equity, Diversity and Inclusion
    - SEC / Foundational Security
- **Required skill level:** Apply, Develop
- **Core level:**

- **Task 8:** Create a database for a new application.
- **Competency Statement:** Design the data storage needs (data modeling), assess the social and ethical implications for collecting and storing the data, determine how to store a new application's data (RDBMS vs NoSQL), and create the database, including appropriate indices.
- **Competency area:** Design, Software, Systems
- **Competency unit:** Development, testing
- **Required knowledge areas and knowledge units:**
    - DM / The Role of Data
    - DM / Core Database Systems Concepts
    - DM / Data Modeling
    - DM / Relational Databases
    - DM / NoSQL Systems
    - DM / DBMS Internals
    - SEP / Social Context
    - SEP / Methods for Ethical Analysis
    - SEP / Privacy and Civil Liberties
    - SEP / Professional Ethics
    - SEP / Security Policies, Laws and Computer Crimes
    - SEP / Equity, Diversity and Inclusion

- SEC / Foundational Security

- **Required skill level:** Develop
- **Core level:**

Committee

Chair: Mikey Goldweber, Xavier University, Cincinnati, USA

Members:

- Sherif Aly, The American University in Cairo, Cairo, Egypt
- Sara More, Johns Hopkins University, USA
- Mohamed Mokbel, University of Minnesota, USA
- Rajendra Raj, Rochester Institute of Technology, Rochester, USA
- Avi Silberschatz, Yale University, New Haven, USA
- Min Wei, Microsoft
- Qiao Xiang, Xiamen University, China

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
DM	The Role of Data	• The Data Life Cycle • The social/legal aspects of data collections: [crosslist SEP]	Evaluate	CS	2
DM	Core DB	• Purpose and advantages of database systems • Components of database systems • Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods) • Database architecture, data independence, and data abstraction			

	System Concepts	● Transaction mgmt ● Normalization ● Approaches for managing large volumes of data (e.g., noSQL database systems, use of MapReduce). [crosslist PD] ● Distributed databases/cloud-based systems ● Structured, semi-structured, and unstructured databases	Explain	CS	2
		● Use of a declarative query language	Develop		
DM	Core DB System Concepts	● Systems supporting structured and/or stream content	Explain	KA	.5
DM	Data Modeling	● Data modeling ● Relational data models	Develop	CS	2
DM	Data Modeling	● Conceptual models (e.g., entity-relationship, UML diagrams) ● Semi-structured data model (expressed using DTD, XML, or JSON Schema, for example)	Explain	KA	3
DM	Relational Databases	● Entity and referential integrity ○ Candidate key, superkeys ● Relational database design	Explain	CS	1
DM	Relational Databases	● Mapping conceptual schema to a relational schema ● Physical database design: file and storage structures ● Functional dependency Theory ● Normalization Theory ○ Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition Normal forms (BCNF) ○ Denormalization (for efficiency)	Develop	KA	3

DM	Query Construction	● SQL Query Formulation	Develop	CS	2

DM	Query Construction	● Relational Algebra
		● SQL
		  ○ Data definition including integrity and other constraints specification
		  ○ Update sublanguage
			Develop	KA	4

DM	Query Processing	● Index structures
		  ○ B+ trees
		  ○ Hash indices: static and dynamic
		  ○ Index creation in SQL
		● Algorithms for query operators
		  ○ External Sorting
		  ○ Selection
		  ○ Projection; with and without duplicate elimination
		  ○ Natural Joins: Nested loop, Sort-merge, Hash join
		  ○ Analysis of algorithm efficiency
		● Query transformations
		● Query optimization
		  ○ Access paths
		  ○ Query plan construction
		  ○ Selectivity estimation
		  ○ Index-only plans
			Explain	KA	4

		● Database tuning: Index selection ○ Impact of indices on query performance	Develop		

DM	DBMS Internals	● DB Buffer Management
		● Transaction Processing
		  ○ Isolation Levels
		  ○ ACID
		  ○ Serializability
		● Concurrency Control: [crosslist PD]
		  ○ 2-Phase Locking
		  ○ Deadlocks handling strategies
		● Recovery Manager
		  ○ Relation with Buffer Manager
			Explain	KA	4

DM	NoSQL System	● Why NoSQL? (e.g. Impedance mismatch between Application [CRUD] and RDBMS) ● Key-Value and Document data model	Explain	KA	2

DM	Data Analytics	tbd			
DM	Data Security & Privacy	tbd			

Foundations of Programming Languages (FPL)

Preamble

This knowledge area provides a basis (rooted in discrete mathematics and logic) for the understanding of complex modern programming languages: their foundations, implementation, and formal description. Although programming languages vary according to the language paradigm and the problem domain, and evolve in response to both societal needs and technological advancement, they share an underlying abstract model of computation and program development. This remains true even as processor hardware and their interface with programming tools are becoming increasingly intertwined and progressively more complex. An understanding of the common abstractions and programming paradigms enables faster learning of new programming languages.

The Foundations of Programming Languages Knowledge Area is concerned with articulating the underlying concepts and principles of programming languages, the formal specification of a programming language and the behavior of a program, explaining how programming languages are implemented, comparing the strengths and weaknesses of various programming paradigms, and describing how programming languages interface with entities such as operating systems and hardware. The concepts covered in this area are applicable to many different languages and an understanding of these principles assists in being able to move readily from one language to the next, and to be able to select a programming paradigm and a programming language to best suit the problem at hand.

Two example courses are presented at the end of this knowledge area to illustrate how the content may be covered. The first is an introductory course which covers the CS Core and KA Core content. This is a course focused on the different programming paradigms and ensue familiarity with each to a level sufficient to be able to decide which paradigm is appropriate in which circumstances.

The second course is an advanced course focused on the implementation of a programming language and the formal description of a programming language and a formal description of the behavior of a program.

While these two courses have been the predominant way to cover this knowledge area of the past decade, it is by no means the only way that the content can be covered. An institution could,

for example, choose to cover only the CS Core content (28 hours) in a shorter course, or in a course which combines this CS Core content with Core content from another knowledge area such as Software Engineering. Natural combinations are easily identifiable since they are the areas in which the Foundations of Programming Languages knowledge areas overlaps with other knowledge areas. A list of such overlap areas is provided at the end of this knowledge area.

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. Over the course of a career, a computer scientist will need to learn and work with many different languages, separately or together. Software developers must understand the programming models, new programming features and constructs, 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 to improve execution efficiency and long-term maintenance of developed software. The effective use of programming languages and appreciation of their limitations also requires a basic knowledge of programming language translation and program analysis, of run-time behavior, and components such as memory management and the interplay of concurrent processes communicating with each other through message-passing, shared memory, and synchronization, Finally, some developers and researchers will need to design new languages, an exercise which requires familiarity with basic principles.

Changes since CS 2013: These include a change in name of the Knowledge Area from Programming Languages to Foundations of Programming Languages to better reflect the fact that the KA is about the fundamentals underpinning programming languages and related concepts, and not about any specific programming languages. Changes also include a redistribution of content formerly identified as core tier-1 and core tier-2 within the Programming Language Knowledge Area (KA). These are now CS Core hours and KA Core hours. All computer science graduates are expected to have the CS Core hours, and those graduates that specialize in a knowledge area are also expected to have the KA core hours. Content that is not identified as either CS Core hours or KA Core hours are non-core topics. The variation in core hours from 2013 reflects the change in importance or relevance of topics over the past decade. The inclusion of new topics is driven by their current prominence in the programming language landscape, or the anticipated impact of emerging areas on the profession in general. Specifically, the changes are:

- Object-Oriented Programming +1 CS Core hours, -2 KA Core hours
- Functional Programming +1 CS Core hour
- Logic Programming +3 CS Core hours
- Event-Driven and Reactive Programming +2 CS Core hours
- Parallel and Distributed Computing +1 CS Core hours, +2 KA Core hours
- Type Systems +1 CS Core hours, -1 KA Core hours
- Language Translation and Execution +4 CS Core hours, -3 KA Core hours

- Program Representation +2 KA Core hours

In addition, some knowledge units are renamed to more accurately reflect their content:

- Static Analysis is renamed to Program Analysis and Analyzers
- Concurrency and Parallelism is renamed to Parallel and Distributed Computing
- Runtime Systems is renamed to Runtime Behavior and Systems
- Basic Type Systems and Type Systems were merged into a single topic and named Type Systems

Four new knowledge units have been added to reflect their continuing and growing importance as we look toward the 2030s:

- Scripting +2 CS Core hours
- Formal Development Methodologies
- Embedded Systems and Hardware Interface
- Fundamentals of Programming Languages and Society, Ethics and Professionalism

Note:

- Some topics from one or more of the first three Knowledge Units (Object-Oriented Programming, Functional Programming, Event-Driven and Reactive Programming) are likely to be integrated with topics in the Software Development Fundamentals Knowledge Area in a curriculum's introductory courses. Curricula will differ on which topics are integrated in this fashion and which are delayed until later courses on software development and programming languages.
- Different programming paradigms correspond to different problem domains. Most languages have evolved to integrate more than one programming paradigms such imperative with OOP, functional programming with OOP, logic programming with OOP, and event and reactive modeling with OOP. Hence, the emphasis is not on just one programming paradigm but a balance of all major programming paradigms.
- With multicore computing, cloud computing, and computer networking becoming commonly available in the market, it has become imperative to understand the integration of "Distribution, concurrency, parallelism, and code mobility/migration" along with other programming paradigms as a core area. This paradigm is integrated with almost all other major programming paradigms.
- With the recent increased emphasis on data management, and artificial intelligence, concepts suitable for data management and AI programming have become imperative, New knowledge areas "Data Management" and "Artificial Intelligence" have been introduced. Hence, predicate-based database, and AI programming present in the logic programming paradigm, has been introduced as a core CS area along with functional programming.

- With ubiquitous computing and real-time temporal computing becoming more in daily human life such as health, transportation, smart homes, it has become important to cover the software development aspect of “Embedded Computing and Hardware Interfaces” under programming languages. Some of the topics covered will require, and interface with, concepts covered in knowledge areas such as “Architecture and Organization”, “Operating Systems”, and “Systems Fundamentals”.
- Some topics from the Parallel and Distributed Computing Knowledge Unit are likely to be integrated within the curriculum with topics from the Parallel and Distributed Programming Knowledge Area.
- Some topics from the Hardware Interface Knowledge Unit are likely to integrated within the curriculum with topics from the System Fundamentals Knowledge Area.

Core Hours

Knowledge Units	CS Core	KA Core
Object-Oriented Programming	6	3
Functional Programming	4	4
Logic Programming		3
Scripting	2	
Event-Driven and Reactive Programming	2	2
Parallel and Distributed Computing	3	2
Type Systems	3	4
Language Translation and Execution	4	
Program Representation		3
Syntax Analysis		
Compiler Semantic Analysis		
Program Analysis and Analyzers		
Code Generation		
Runtime Behavior and Systems		
Advanced Programming Constructs		
Language Pragmatics		
Formal Semantics		
Formal Development Methodologies		
Embedded Computing and Hardware Interface		
FPL and SEP		
Total	**24**	**21**

Knowledge Units

FPL/Object-Oriented Programming

[6 CS Core hours, 3 KA Core hours]

- Topics
  - [CS Core]
    - Object-oriented design
      - Decomposition into objects carrying state and having behavior
      - Class-hierarchy design for modeling
    - Definition of classes: fields, methods, and constructors
    - Subclasses, inheritance (including multiple inheritance), and method overriding
    - Dynamic dispatch: definition of method-call
    - Exception handling
    - Object-oriented idioms for encapsulation
      - Privacy, data hiding, and visibility of class members
      - Interfaces revealing only method signatures
      - Abstract base classes, traits and mixins
    - Dynamic vs static properties
    - Composition vs inheritance

  - [KA Core]
    - Subtyping
      - Subtype polymorphism; implicit upcasts in typed languages
      - Notion of behavioral replacement: subtypes acting like supertypes
      - Relationship between subtyping and inheritance
      - Collection classes, iterators, and other common library components

- Illustrative learning outcomes
  - [CS Core]
    1. Compose a class through design, implementation, and testing to meet behavioral requirements. [Creating]
    2. Build a simple class hierarchy utilizing subclassing that allows code to be reused for distinct subclasses. [Applying]
    3. Predict and validate control flow in a program using dynamic dispatch. [Analyzing, Evaluating]
    4. Compare and contrast
       a) the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant-and
       b) 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. [Analyzing] [*cross-reference: FPL/Functional Programming*]

5. Compare and contrast the benefits and costs/impact of using inheritance (subclasses) and composition (in particular how to base composition on higher order functions). [Analyzing]

- [KA Core]
  1. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype).
  2. Use object-oriented encapsulation mechanisms such as interfaces and private members. [Applying]
  3. 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. [Enumerate, Applying] [*cross-reference: FPL/Functional Programming*]

FPL/Functional Programming

[4 CS Core hours, 4 KA Core hours]

- Topics
  - [CS Core]
    - Lambda expressions and evaluation
      - Variable binding and scope rules
      - Parameter passing
      - Nested lambda expressions and reduction order
    - Effect-free programming
      - Function calls have no side effects, facilitating compositional reasoning
      - Immutable variables and data copying vs. reduction
      - Use of recursion vs. loops vs. pipelining (map/reduce)
    - Processing structured data (e.g., trees) via functions with cases for each data variant
      - Functions defined over compound data in terms of functions applied to the constituent pieces
      - Persistent data structures
    - Using higher-order functions (taking, returning, and storing functions)

  - [KA Core]
    - Function closures (functions using variables in the enclosing lexical environment)
      - Basic meaning and definition - creating closures at run-time by capturing the environment

        - Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
        - Using a closure to encapsulate data in its environment
        - Lazy versus eager evaluation
    - Defining and implementing functions which can accept a function as a parameter, combined with another function, or return a function.

- [Non-Core]
    - Graph reduction machine and call-by-need
    - Implementing lazy evaluation
    - Integration with logic programming paradigm using concepts such as equational logic, narrowing, residuation and semantic unification [*cross-reference: FPL/Logic Programming*]
    - Integration with other programming paradigm such as imperative and object-oriented

- Illustrative learning outcomes
    - [CS Core]
        1. Develop basic algorithms that avoid assigning to mutable state or considering reference equality. [Creating]
        2. Develop useful functions that take and return other functions. [Creating]
        3. Compare and contrast
            a) the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant, and
            b) 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. [Analyzing]

            [*cross-reference: FPL/Object-Oriented Programming*]

    - [KA Core]
        4. Explain a simple example of lambda expression being implemented using SECD machine showing storage and reclaim of the environment
        5. Correctly interpret variables and lexical scope in a program using function closures.
        6. Use functional encapsulation mechanisms such as closures and modular interfaces. [Applying]
        7. Compare and contrast stateful vs stateless execution [Analyzing]
        8. 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. [Enumerate, Applying]

[*cross-reference: FPL/Object-Oriented Programming*]

- [Non-core]
  9. Illustrate graph reduction using a λ-expression using a shared subexpression
  10. Illustrate the execution of a simple nested λ-expression using an ABC machine
  11. Illustrate narrowing, residuation and semantic unification using simple illustrative examples
  12. Illustrate the concurrency constructs using simple programming examples of known concepts such as a buffer being read and written concurrently or sequentially

FPL/Logic Programming

[3 KA Core hours, Non-core]

- Topics
  - [KA Core}
    - Universal vs. existential quantifiers
    - First order predicate logic vs. higher order logic
    - Expressing complex relations using logical connectives and simpler relations
    - Definitions of Horn clause, facts, goals and subgoals
    - Unification and unification algorithm; unification vs. assertion vs expression evaluation
    - Mixing relations with functions
    - Cuts, backtracking and non-determinism
    - Closed-world vs. open-world assumptions

  - [Non-Core]
    - Memory overhead of variable copying in handling iterative programs
    - Programming constructs to store partial computation and pruning search trees
    - Mixing functional programming and logic programming using concepts such as equational logic, narrowing, residuation and semantic unification [*cross-reference FPL/Functional Programming*]
    - Higher-order, constraint and inductive logic programming
    - Integration with other programming paradigms such as object-oriented programming
    - Advance programming constructs such as difference-lists, creating user defined data structures, set of, etc.

- Illustrative Learning Outcomes

- [KA Core]
  1. Use a logic language to implement a conventional algorithm. [Applying]
  2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts. [Applying]
  3. Use a simple illustrative example to show correspondence between First Order predicate Logic (FOPL) and logic programs using Horn clauses [Applying]
  4. Use examples to illustrate unification algorithm and its role of parameter passing in query reduction [Applying]
  5. Use simple logic programs interleaving relations, functions, and recursive programming such as factorial and Fibonacci numbers and simple complex relationships between entities, and illustrate execution and parameter passing using unification and backtracking [Applying]

- [Non-core]
  6. Illustrate computation of simple programs such as Fibonacci and show overhead of recomputation, and then show how to improve execution overhead

FPL/Scripting

[2 CS Core hours] [*cross-reference: SDF and OS*]

- Topics
  - Divide, combine, conquer
  - Concurrency
  - Error/exception handling
  - I/O redirection
  - System commands
  - Environment variables
  - File test operators
  - Data structures
    - Arrays and lists
    - Slices
    - List Comprehensions
  - Regular expressions
  - Dynamic typing
  - Function declarations
  - Processes and threads
  - Code objects

- Illustrative Learning Outcomes
  1. Create and execute automated scripts to manage various system tasks. [Applying]
  2. Solve various text processing problems through scripting [Applying]

FPL/Event-Driven and Reactive Programming

[2 CS Core hour, 2 KA Core hours]

This material can stand alone or be integrated with other knowledge units on concurrency, asynchrony, and threading to allow contrasting events with threads.

- Topics
  - [CS Core]
    - Procedural programming vs. reactive programming: advantages of reactive programming in capturing events
    - Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers
    - Behavior model of event-based programming
    - Canonical uses such as GUIs, mobile devices, robots, servers
    - Reactive programs as state transition system

  - [KA Core]
    - Using a reactive framework
      - Defining event handlers/listeners
      - Parameterization of event senders and event arguments
      - Externally-generated events and program-generated events
    - Separation of model, view, and controller

- Illustrative Learning Outcomes
  - [CS Core]
    1. Implement event handlers for use in reactive systems, such as GUIs. [Applying]
    2. Examine why an event-driven programming style is natural in domains where programs react to external events. [Analyzing]

  - [KA Core]
    3. Define and use a reactive framework [Using]
    4. Describe an interactive system in terms of a model, a view, and a controller.

FPL/Parallel and Distributed Computing

[3 CS Core hours, 2 KA Core Hours]

Support for concurrency is a fundamental programming-languages issue with rich material in programming language design, language implementation, and language theory. Due to coverage in other Knowledge Areas, this non-core Knowledge Unit aims only to complement the material

included elsewhere in the body of knowledge. Courses on programming languages are an excellent place to include a general treatment of concurrency including this other material.

[*Cross-reference: PD/Parallel and Distributed Computing, SF/Parallelism*]

- Topics
  - [CS Core]
    - Safety and liveness
      - Race conditions
      - Dependencies/preconditions
      - Fault models
      - Termination
    - Programming models
      - Actor models
      - Procedural and reactive models
      - Synchronous/asynchronous programming models
      - Data parallelism
    - Semantics
      - Commutativity
      - Ordering
      - Independence
      - Consistency
      - Atomicity
      - Consensus
    - Execution control
      - Async await
      - Promises
      - Threads
    - Communication and coordination
      - Message-passing
      - Shared memory
      - cobegin-coend
      - Monitors
      - Channels
      - Threads
      - Guards

  - [KA Core]
    - Futures
    - Language support for data parallelism such as forall, loop unrolling, map/reduce
    - Effect of memory-consistency models on language semantics and correct code generation
    - Representational State Transfer Application Programming Interfaces (REST APIs)

- Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing
- Overheads of message passing
- Granularity of program for efficient exploitation of concurrency.
- Concurrency and other programming paradigms (e.g., functional).

- Illustrative Learning Outcomes
  - [CS Core]
    1. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result.
    2. Implement correct concurrent programs using multiple programming models, such as shared memory, actors, futures, synchronization constructs, and data-parallelism primitives. [Applying]
    3. Use a message-passing model to analyze a communication protocol. [Applying]
    4. Use synchronization constructions such as monitor/synchronized methods in a simple program [Applying]
    5. Modeling data dependency using simple programming constructs involving variables, read and write [Applying]
    6. Modeling control dependency using simple constructs such as selection and iteration [Applying]

  - [KA Core]
    7. Explain how REST API's integrate applications and automate processes .
    8. Explain benefits, constraints and challenges related to distributed and parallel computing .

FPL/Type Systems

[3 CS Core hours, 4 KA Core hours, Non-core]

The KA Core hours would be profitably spent both on the KA Core topics and on a less shallow treatment of the CS Core topics and learning outcomes.

- Topics
  - [CS Core]
    - A type as a set of values together with a set of operations
      - Primitive types (e.g., numbers, Booleans)
      - Compound types built from other types (e.g., records, unions, arrays, lists, functions, references) using set operations
    - Association of types to variables, arguments, results, and fields
    - Type safety as an aspect of program correctness [*cross-reference: FPL/Formal Semantics*]
    - Type safety and errors caused by using values inconsistently given their intended types

- Statically-typed vs dynamically-typed programming languages
- Type equivalence: structural vs name equivalence
- Goals and limitations of static and dynamic typing
  - Detecting and eliminating errors as early as possible
- Generic types (parametric polymorphism) [*cross-reference: FPL/Formal Semantics*]
  - Definition and advantages of polymorphism: parametric, subtyping, overloading and coercion
  - Comparison of monomorphic and polymorphic types
  - Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism
  - Generic parameters and typing
  - Use of generic libraries such as collections
  - Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
  - Prescriptive vs. descriptive polymorphism
  - Implementation models of polymorphic types

- [KA Core]
  - Complementary benefits of static and dynamic typing
    - Errors early vs. errors late/avoided
    - Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections
    - Typing rules
      - Rules for function, product, and sum types
    - Alternative type systems such as Hindley-Miller and Grant-Reynolds type systems.
    - Avoid misuse of code vs. allow more code reuse
    - Detect incomplete programs vs. allow incomplete programs to run
    - Relationship to static analysis
    - Decidability
    - Use of sophisticated (complex) type systems, e.g., Rust.

- [Non-core]
  - Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
  - Type checking
  - Subtyping [*cross-reference: FPL/Object-Oriented Programming*]
    - Subtype polymorphism; implicit upcasts in typed languages
    - Notion of behavioral replacement: subtypes acting like supertypes
    - Relationship between subtyping and inheritance

- Type safety as preservation plus progress
- Type inference
- Static overloading
- Propositions as types (implication as a function, conjunction as a product, disjunction as a sum) [*cross-reference: FPL/Formal Methods*]
- Dependent types (universal quantification as dependent function, existential quantification as dependent product) [*cross-reference: FPL/Formal Methods*]

- Illustrative Learning Outcomes
  - [CS Core]
    1. Describe, for both a primitive and a compound type, the values that have that type.
    2. Describe, for a language with a static type system, the operations that are forbidden statically, such as passing the wrong type of value to a function or method.
    3. Describe examples of program errors detected by a type system.
    4. Identify program properties, for multiple programming languages, that are checked statically and program properties that are checked dynamically. [Enumerate]
    5. Describe an example program that does not type-check in a particular language and yet would have no error if run.
    6. Use types and type-error messages to write and debug programs. [Applying]

  - [KA Core]
    7. Explain how typing rules define the set of operations that are legal for a type.
    8. List the type rules governing the use of a particular compound type. [Enumerate]
    9. Explain why undecidability requires type systems to conservatively approximate program behavior.
    10. Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections. [Enumerate]
    11. Discuss the differences among generics, subtyping, and overloading.
    12. Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software.

  - [Non-core]
    13. Define a type system precisely and compositionally. [Enumerate]
    14. For various foundational type constructors, identify the values they describe and the invariants they enforce. [Enumerate]
    15. Precisely describe the invariants preserved by a sound type system.
    16. Prove type safety for a simple language in terms of preservation and progress theorems. [Evaluating]
    17. Implement a unification-based type-inference algorithm for a simple language. [Applying]

18. Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs.

FPL/Language Translation and Execution

[4 CS Core hours, Non-core]

- Topics
  - [CS Core]]
    - Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
    - Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution
      - BNF and extended BNF representation of context-free grammar
      - Parse tree using a simple sentence such as arithmetic expression or if-then-else statement
      - Ambiguity in Parsing due to lack of precedence order and resolution
      - Execution as native code or within a virtual machine
      - Alternatives like dynamic loading and dynamic (or "just-in-time") code generation
    - Control-flow diagrams using selection and iteration
    - Data structures for translation, execution, translation and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string
    - Direct, indirect, and indexed access to memory location
    - Runtime representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects
    - Abstract low-level machine with simple instruction, stack and heap to explain translation and execution
    - Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap [*cross reference: AR/Memory System Architecture and Organization, OS/Memory Management*]
      - Translating selection and iterative constructs to control-flow diagrams
      - Translating control-flow diagrams to low level abstract code
      - Implementing loops, recursion, and tail calls
      - Translating function/procedure calls and return from calls, including different parameter passing mechanism using an abstract machine
    - Memory management [*cross reference: OS/Memory Management, FPL/Hardware Interface*]
      - Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects
      - Return from procedure as automatic deallocation mechanism for local data elements in the stack
      - Manual memory management: allocating, de-allocating, and reusing heap memory

- Automated memory management: garbage collection as an automated technique using the notion of reachability

- [Non-core]
  - Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
  - Secure compiler development [*cross-reference: SEC/Foundational Security*]

- Illustrative Learning Outcomes
  [CS Core]

  1. Differentiate a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.).
  2. Differentiate syntax and parsing from semantics and evaluation.
  3. Use BNF and extended BNF to specify the syntax of simple constructs such as if-then-else, type declaration and iterative constructs for known languages such as C++ or Python [Applying]
  4. Illustrate parse tree using a simple sentence/arithmetic expression [Applying]
  5. Illustrate translation of syntax diagrams to BNF/extended BNF for simple constructs such as if-then-else, type declaration, iterative constructs, etc.
  6. Illustrate ambiguity in parsing using nested if-then-else/arithmetic expression and show resolution using precedence order
  7. Diagram a low-level run-time representation of core language constructs, such as data abstractions and control abstractions. [Applying]
  8. Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model.
  9. Investigate, identify, and fix memory leaks and dangling-pointer dereferences. [Applying]

  [Non-core]

  10. Discuss the benefits and limitations of garbage collection, including the notion of reachability.

FPL/Program Representation

[3 KA Core hours]

- Topics
  - BNF and regular expressions
  - Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators

- Components of a language
  - Definitions of alphabets, delimiters, sentences, syntax and semantics
  - Syntax vs. semantics
- Types of semantics: operational, axiomatic, denotational, behavioral; define and use abstract syntax trees; contrast with concrete ssyntaxyntax
- Program as a set of non-ambiguous meaningful sentences
- Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling
- Mutable vs. immutable variables: advantages and disadvantages of reusing existing memory location vs. advantages of copying and keeping old values; storing partial computation vs. recomputation
- L-values and R-values: mapping mutable variable-name to L-values; mapping immutable variable-names to R-values
- Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables
- Scope rules: static vs. dynamic; visibility of variables; side-effects
- Environment vs. store and their properties
- Data and control abstraction
- Mechanisms for information exchange between program units such as procedures, functions and modules: nonlocal variables, global variables, parameter passing, import-export between modules
- Types of parameter passing with simple illustrations and comparison: call by value, call by reference, call by value-result, call by name, call by need and their variations
- Side-effects induced by nonlocal variables, global variables and aliased variables
- Data structures to represent code for execution, translation, or transmission
- Low level instruction representation such as virtual machine instructions, assembly language, and binary representation [*cross-reference: AR/Machine Level Representation of Data*]
- Lambda calculus, variable binding, and variable renaming.
- String-based mobility in mobile coding.

- Illustrative Learning Outcomes
  1. Illustrate the scope of variables and visibility using simple programs
  2. Illustrate different types of parameter passing using simple pseudo programming language
  3. Explain side-effect using global and nonlocal variables and how to fix such programs
  4. Explain how programs that process other programs treat the other programs as their input data.
  5. Describe a grammar and an abstract syntax tree for a small language.
  6. Describe the benefits of having program representations other than strings of source code.

7. Implement a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator. [Applying]

FPL/Syntax Analysis

[Non-core]

- Topics
  - Regular grammars vs. context-free grammars
  - Scanning and parsing based on language specifications
  - Lexical analysis using regular expressions
  - Tokens and their use
  - Parsing strategies including top-down (e.g., recursive descent, or LL) and bottom-up (e.g., LR or GLR) techniques.
    - Lookahead tables and their application to parsing
  - Language theory
    - Chomsky hierarchy
    - Left-most/right-most derivation and ambiguity
    - Grammar transformation
  - Parser error recovery mechanisms
  - Generating scanners and parsers from declarative specifications

- Illustrative Learning Outcomes
  1. Use formal grammars to specify the syntax of languages. [Applying]
  2. Illustrate the role of lookahead tables in parsing
  3. Use declarative tools to generate parsers and scanners. [Applying]
  4. Recognize key issues in syntax definitions: ambiguity, associativity, precedence. [Enumerate]

FPL/Compiler Semantic Analysis

[Non-core]

- Topics
  - Abstract syntax trees; contrast with concrete syntax
  - Defining, traversing and modifying high-level program representations
  - Scope and binding resolution
  - Static semantics
    - Type checking
    - Define before use
    - Annotation and extended static checking frameworks
  - L-values/R-values [*cross reference: SDF/Fundamental Programming Constructs*]
  - Call semantics
  - Parameter passing mechanisms

- Declarative specifications such as attribute grammars and their applications in handling limited context-base grammar

- Illustrative Learning Outcomes
  1. Describe an abstract syntax tree for a small language
  2. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences. [Applying]
  3. Describe semantic analyses using an attribute grammar.

FPL/Program Analysis and Analyzers

[Non-core]

- Topics
  - Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, and static single assignment.
  - Undecidability and consequences for program analysis
  - Flow-insensitive analysis, such as type-checking and scalable pointer and alias analysis
  - Flow-sensitive analysis, such as forward and backward dataflow analyses
  - Path-sensitive analysis, such as software model checking and software verification
  - Tools and frameworks for implementing analyzers
  - Role of static analysis in program optimization and data dependency analysis during exploitation of concurrency [cross-reference: *FPL/Code Generation*]
  - Role of program analysis in (partial) verification and bug-finding [*cross reference: FPL/Code Generation*]
  - Parallelization
    - Analysis for auto-parallelization
    - Analysis for detecting concurrency bugs

- Illustrative Learning Outcomes
  1. Define useful program analyses in terms of a conceptual framework such as dataflow analysis. [Enumerate]
  2. Explain the difference between dataflow graph and control flow graph.
  3. Explain why non-trivial sound program analyses must be approximate.
  4. Argue why an analysis is correct (sound and terminating). [Analyzing]
  5. Explain why potential aliasing limits sound program analysis and how alias analysis can help.
  6. Use the results of a program analysis for program optimization and/or partial program correctness. [Applying]

FPL/Code Generation

[Non-core]

- Topics
  - Instruction sets [*cross reference: AR/Assembly Level Machine Organization*]
  - Control flow
  - Memory management [*cross reference: AR/Memory System Organization and Architecture, OS/Memory Management*]
  - Procedure calls and method dispatching
  - Separate compilation; linking
  - Instruction selection
  - Instruction scheduling (e.g., pipelining)
  - Register allocation
  - Code optimization as a form of program analysis [*cross reference: FPL/Program Analyzers*]

- Illustrative Learning Outcomes
  1. Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Enumerate]
  2. Generate the low-level code for calling functions/methods in modern languages. [Creating]
  3. Discuss why separate compilation requires uniform calling conventions.
  4. Discuss why separate compilation limits optimization because of unknown effects of calls.
  5. Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization.

FPL/Runtime Behavior and Systems

[Non-core]

- Topics
  - Process models using stacks and heaps to allocate and deallocate activation records and recovering environment using frame pointers and return addresses during a procedure call including parameter passing examples.
  - Schematics of code lookup using hash tables for methods in implementations of object-oriented programs
  - Data layout for objects and activation records
  - Object allocation in heap
  - Implementing virtual entities and virtual methods; virtual method tables and their application
  - Runtime behavior of object-oriented programs

- Compare and contrast allocation of memory during information exchange using parameter passing and non-local variables (using chain of static links)
- Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
- Just-in-time compilation and dynamic recompilation
- Interface to operating system (e.g., for program initialization)
- Interoperability between programming languages including parameter passing mechanisms and data representation
  - Big Endian, little endian
  - Data layout of composite data types such as arrays
- Other common features of virtual machines, such as class loading, threads, and security checking
- Sandboxing

- Illustrative Learning Outcomes
  1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Analyzing]
  2. Discuss benefits and limitations of automatic memory management.
  3. Explain the use of metadata in run-time representations of objects and activation records, such as class pointers, array lengths, return addresses, and frame pointers.
  4. Compare and contrast static allocation vs. stack-based allocation vs. heap-based allocation of data elements. [Analyzing]
  5. Explain why some data elements cannot be automatically deallocated at the end of a procedure/method call (need for garbage collection).
  6. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation.
  7. Discuss use of sandboxing in mobile code
  8. Identify the services provided by modern language run-time systems. [Enumerate]

FPL/Advanced Programming Constructs

[Non-core]

- Topics
  - Encapsulation mechanisms
  - Lazy evaluation and infinite streams
  - Compare and contrast lazy evaluation vs. eager evaluation
  - Unification vs. assertion vs. expression evaluation
  - Control Abstractions: Exception Handling, Continuations, Monads
  - Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
  - Metaprogramming: Macros, Generative programming, Model-based development
  - String manipulation via pattern-matching (regular expressions)

- Dynamic code evaluation ("eval")
- Language support for checking assertions, invariants, and pre/post-conditions
- Domain specific languages, such as database languages, data science languages, embedded computing languages, synchronous languages, hardware interface languages
- Massive parallel high performance computing models and languages

- Illustrative Learning Outcomes
  1. Use various advanced programming constructs and idioms correctly.
  2. Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]
  3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]

FPL/Language Pragmatics

[Non-core]

- Topics
  - Effect of technology needs and software requirements on programming language development and evolution
  - Problems domains and programming paradigm
    - Criteria for good programming language design
    - Principles of language design such as orthogonality
    - Defining control and iteration constructs
    - Modularization of large software
  - Evaluation order, precedence, and associativity
  - Eager vs. delayed evaluation
  - Defining control and iteration constructs
  - External calls and system libraries

- Illustrative Learning Outcomes
  1. Discuss the role of concepts such as orthogonality and well-chosen defaults in language design.
  2. Use crisp and objective criteria for evaluating language-design decisions. [Applying]
  3. Implement an example program whose result can differ under different rules for evaluation order, precedence, or associativity. [Applying]
  4. Illustrate uses of delayed evaluation, such as user-defined control abstractions. [Applying]
  5. Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation.

FPL/Formal Semantics

[*cross-reference: SE/Formal Methods*]

[Non-core]

- Topics
  - Syntax vs. semantics
  - Approaches to semantics: Axiomatic, Operational, Denotational, Type-based,
  - Axiomatic semantics of abstract constructs such as assignment, selection, iteration using pre-condition, post-conditions and loop invariance
  - Operational semantics analysis of abstract constructs and sequence of such as assignment, expression evaluation, selection, iteration using environment and store
    - Symbolic execution
    - Constraint checkers
  - Denotational semantics
    - Lambda Calculus
  - Proofs by induction over language semantics
  - Formal definitions and proofs for type systems [*cross-reference: FPL/Type Systems*]
    - Propositions as types (implication as a function, conjunction as a product, disjunction as a sum)
    - Dependent types (universal quantification as dependent function, existential quantification as dependent product)
    - Parametricity

- Illustrative Learning Outcomes
  1. Construct a formal semantics for a small language. [Creating]
  2. Write a lambda-calculus program and show its evaluation to a normal form. [Applying]
  3. Discuss the different approaches of operational, denotational, and axiomatic semantics.
  4. Use induction to prove properties of all programs in a language. [Applying]
  5. Use induction to prove properties of all programs in a language that are well-typed according to a formally defined type system. [Applying]
  6. Use parametricity to establish the behavior of code given only its type. [Applying]

FPL/Formal Development Methodologies

[*cross-reference: SE/Formal Methods*]

[Non-core]

- Topics
  - Formal specification languages and methodologies
  - Theorem provers, proof assistants, and logics

- Constraint checkers [*cross-reference: FPL/Formal Methods*]
- Dependent types (universal quantification as dependent function, existential quantification as dependent product) [*cross-reference: FPL/Type Systems and FPL/Formal Methods*]
- Specification and proof discharge for fully verified software systems using pre/post conditions, refinement types, etc.
- Formal modelling and manual refinement/implementation of software systems
- Use of symbolic testing and fuzzing in software development
- Model checking
- Understanding of situations where formal methods can be effectively applied and how to structure development to maximize their value.

- Illustrative Learning Outcomes
  1. Use formal modeling techniques to develop and validate architectures. [Applying]
  2. Use proof assisted programming languages to develop fully specified and verified software artifacts. [Applying]
  3. Use verifier and specification support in programming languages to formally validate system properties. [Applying]
  4. Integrate symbolic validation tooling into a programming workflow. [Analyzing]
  5. Discuss when and how formal methods can be effectively used in the development process.

FPL/Embedded Computing and Hardware Interface

[Non-core]

- Topics
  - Languages for custom architectures
    - GPU technology
    - Field Programmable Gate Arrays
    - Languages for heterogenous systems
  - Embedded Systems
    - Microcontrollers
    - Interrupts and feedback
    - Interrupt handlers in high level languages
      - Hard and soft interrupts and trap-exits
    - Interacting with hardware, actuators, and sensors
    - Energy efficiency
    - Loosely timed coding and synchronization
    - Software adapters
  - Real-time systems
    - Hard real-time systems vs soft real-time systems

    - Timeliness
    - Time synchronization/scheduling
    - Prioritization
    - Latency
    - Compute jitter
  - Memory management [*cross-reference: FPL/Language Translation and Execution*]
    - Mapping programming construct (variable) to a memory location
    - Shared memory
    - Manual memory management
    - Garbage collection

- Illustrative Learning Outcomes
  1. Design and develop software to interact with and control hardware [Creating]
  2. Design methods for real-time systems [Creating]
  3. Evaluate real-time scheduling and schedulability analysis [Analyzing]
  4. Evaluate formal specification and verification of timing constraints and properties [Analyzing]

FPL/FPL and SEP

[Non-core]

- Topics
  - Impact of English-centric programming languages
  - Enhancing accessibility and inclusivity for people with disabilities
    - Supporting assistive technologies
  - Human factors related to programming languages and usability
    - Impact of syntax on accessibility
    - Supporting cultural differences (e.g., currency, decimals, dates)
    - Neurodiversity
  - Epistemology of terms such as “class”, “master”, “slave” in programming languages
  - Increasing accessibility by supporting multiple languages within applications (UTF)

- Illustrative Learning Outcomes
  1. Consciously design programming languages to be inclusive and non-offensive [Applying]

Professional Dispositions

- *Professional*: Students must demonstrate the highest professional standards when using programming languages and formal methods to build safe systems that are fit for their purpose.
- *Inventive*: Programming and approaches to formal proofs is inherently a creative process, students must demonstration innovative approaches to problem solving.
- *Meticulous*: Attention to detail is essential when using programming languages and applying formal methods.
- *Responsible*: Programmers are responsible for anticipating all forms of user input and system behavior and to design solutions that address each one.
- *Perseverance*: Students must demonstrate perseverance since the correct approach is not always self-evident and a process of refinement may be necessary to reach the solution.
- *Accountable*: Students are accountable for their choices regarding the way a problem is solved.

Math Requirements

Needed:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).

Shared Concepts and Crosscutting Themes

Shared Concepts:

- *FPL/Event-Driven and Reactive Programming* overlaps with *AL/Automata and Complexity*
- *FPL/Distribution, Concurrency and Parallelism* overlaps with *PD/Parallel and Distributed Computing, SF/Parallelism*
- *FPL/Program Representation* overlaps with *AR/Machine Level Representation of Data*
- *FPL/Language Translation and Execution* overlaps with *AR/Memory System Architecture and Organization, OS/Memory Management, SEC/Foundational Security*
- *FPL/Compiler Semantic Analysis* overlaps with *SDF/Fundamental Programming Constructs*
- *FPL/Code Generation* overlaps with AR/Assembly Level Machine Organization, AR/Memory System Organization and Architecture, OS/Memory Management
- *FPL/Formal Semantics* overlaps with *SE/Formal Methods*
- *FPL/Formal Development Methodologies* overlaps with *SE/Formal Methods*

- *FPL/Program Analysis and Analyzers* overlaps with *PD/Parallelism Fundamentals*

Course Packaging Suggestions

Introductory Course to include the following:

- FPL/Object-Oriented Programming — 5 CS Core hours, 3 KA Core hours
- FPL/Functional Programming — 4 CS Core hours, 4 KA Core hours
- FPL/Logic Programming — 3 CS Core hours
- FPL/Event-Driven and Reactive Programming — 2 CS Core hours, 2 KA Core hours
- FPL/Scripting — 2 CS Core hours
- FPL/Parallel and Distributed Computing — 3 CS Core hours, 2 KA Core hours
- FPL/Type Systems — 3 CS Core hours, 4 KA Core hours
- FPL/Language Translation and Execution — 4 CS Core hours
- FPL/Program Representation — 3 KA Core hours
- FPL/Advance Programming Constructs — 4 Non-core hours
- FPL/Hardware Interface — 2 Non-core hours
- FPL/FPL and SEP — 1 Non-Core hour

Pre-requisites:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar

Advanced Course to include the following:

- FPL/Language Translation and Execution — 3 Non-core hours
- FPL/Syntax Analysis — 3 Non-core hours
- FPL/Type Systems — 3 Non-core hours
- FPL Compiler Semantic Analysis — 5 Non-core hours
- FPL/Program Analyzers — 5 Non-core hours
- FPL/Code Generation — 5 Non-core hours
- FPL/Runtime Systems — 4 Non-core hours
- FPL/Language Pragmatics — 3 Non-core hours
- FPL/Hardware Interface — 3 Non-core hours
- FPL/Formal Semantics — 5 Non-core hours
- FPL/Formal Development Methodologies — 5 Non-core hours

Pre-requisites:

- Discrete mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar

- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).
- Introductory course.
- Programming proficiency in programming concepts such as:
  - type declarations such as basic data types, records, indexed data elements such as arrays and vectors, and class/subclass declarations, types of variables,
  - scope rules of variables,
  - selection and iteration concepts, function and procedure calls, methods, object creation
- Data structure concepts such as:
  - abstract data types, sequence and string, stack, queues, trees, dictionaries
  - pointer-based data structures such as linked lists, trees and shared memory locations
  - Hashing and hash tables
- System fundamentals and computer architecture concepts such as:
  - Digital circuits design, clocks, bus
  - registers, cache, RAM and secondary memory
  - CPU and GPU
- Basic knowledge of operating system concepts such as:
  - Interrupts, threads and interrupt-based/thread-based programming
  - Scheduling, including prioritization
  - Memory fragmentation
  - Latency

Competency Specifications

- **Task 1:** Make an informed decision regarding which programming language/paradigm to select and use for a specific application.
- **Competency area:** Software/Application
- **Competency unit:** Design/Development/Evaluation
- **Competency Statement:** Apply knowledge of multiple programming paradigms, including their strengths and weaknesses relative to the application to be developed, and select an appropriate paradigm and programming language.
- **Required knowledge areas and knowledge units:**
  - FPL/Object-Oriented Programming
  - FPL/Functional Programming
  - FPL/Logic Programming
  - FPL/Event-Driven and Reactive programming
  - FPL/Type Systems
  - FPL Language Translation and Execution
  - FPL/Language Pragmatics
  - FPL/Embedded Computing and Hardware Interface
  - FPL/Advanced Programming Constructs

- **Required skill level:** Explain/Evaluate
- **Core level:**

- **Task 2:** Use a scripting language to perform or automate a repetitive task..
- **Competency area:** Software/Systems/Application
- **Competency unit:** Development
- **Competency Statement:** Create and execute automated scripts to manage various system and development tasks.
- **Required knowledge areas and knowledge units:**
  - FPL/Scripting
  - OS/File Systems API and Implementation
- **Required skill level:** Develop
- **Core level:**

- **Task 3:** Explain to a co-worker how a reactive or event-driven program works.
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Competency Statement:** Apply knowledge of event-driven and reactive programming to understand and explain the workings of a reactive or event-driven systems..
- **Required knowledge areas and knowledge units:**
  - FPL/Event-Driven and Reactive Programming
  - FPL/Embedded Computing and Hardware Interface
  - SPD/Embedded Platforms
- **Required skill level:** Explain/Evaluate
- **Core level:**

- **Task 4:** Write a white paper which describes the benefits and challenges of a parallel or distributed program.
- **Competency area:** Software/Systems/Application/Theory
- **Competency unit:** Design/Development/
- **Competency Statement:** Apply knowledge of parallel and distributed programming to determine enhancements/benefits over a sequential program.
- **Required knowledge areas and knowledge units:**
  - FPL/Parallel and Distributed Programming
  - PD/Parallel and Distributed Computing
  - SF/Parallelism
- **Required skill level:** Explain/Evaluate
- **Core level:**

- **Task 5:** Write a white paper to describe how a program is translated into machine code and executed.
- **Competency area:** Software/Systems
- **Competency unit:** Evaluation
- **Competency Statement:** Diagram a low-level run-time representation of core language constructs, such as data abstractions and control abstractions. Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model. Be able to investigate, identify, and fix memory leaks and dangling-pointer dereferences.
- **Required knowledge areas and knowledge units:**
  - FPL/Language Translation and Execution
- **Required skill level:** Explain/Evaluate/Apply
- **Core level:**

- **Task 6:** Effectively use a programming language's type system to develop safe and secure software.
- **Competency area:** Software/Application
- **Competency unit:** Development/
- **Competency Statement:** Apply knowledge of static and dynamic type rules for a language to ensure an application is safe, secure, and correct.
- **Required knowledge areas and knowledge units:**
  - FPL/Type Systems
- **Required skill level:** Develop
- **Core level:**

- **Task 7:** Translate input from a high-level language into a lower-level form suitable for use by a computer (e.g., compiler; translate a natural language description in a game into instructions such as function calls).
- **Competency area:** Software/Theory
- **Competency unit:** Design/Development/
- **Competency Statement:** Analyze input, determine its correctness and meaning, and translate into an alternative form appropriate for the application, possibly employing an intermediate form.
- **Required knowledge areas and knowledge units:**
  - FPL/ Language Translation and Execution
  - FPL/Program Representation
  - FPL/Syntax Analysis
  - FPL/Compiler Semantic Analysis
  - FPL/Program Analysis and Analyzers
  - FPL Code Generation

- FPL/Runtime Behavior and Systems
- **Required skill level:** Explain/Apply/Evaluate/Develop
- **Core level:**

- **Task 8:** Discuss a program's correctness relative to a functional specification. **Competency area:** Software/Systems/Application/Theory
- **Competency unit:** Development/Documentation/Acceptance
- **Competency Statement:** Utilize logic and formal methods to argue the correctness of a program.
- **Required knowledge areas and knowledge units:**
  - FPL/Language Pragmatics
  - FPL/Formal Semantics
  - FPL/Formal Development Methodologies
- **Required skill level:** Explain
- **Core level:**

- **Task 9:** Write a program using multiple languages and have the components interact effectively and efficiently with each other.
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Competency Statement:** Apply knowledge of various programming paradigms and languages, and data and their implementation, as well as data representation, to develop a working multi-paradigm solution to a software problem.
- **Required knowledge areas and knowledge units:**
  - FPL/Object-Oriented Programming
  - FPL/Functional Programming
  - FPL/Logic Programming
  - FPL/Event-Driven and Reactive Programming
  - FPL/Advanced Programming Constructs
  - FPL/Type Systems
  - FPL Language Translation and Execution
  - FPL/Language Pragmatics
  - FPL/Embedded Computing and Hardware Interface
  - FPL/Parallel and Distributed Computing
  - FPL/Scripting
  - FPL/Program Representation
  - FPL/Type Systems
  - FPL Language Translation and Execution
- **Required skill level:** Explain/Apply/Evaluate/Develop
- **Core level:**

- **Task 10:** White a white paper explaining how a safe and secure program effectively utilized programming language features to make it safe and secure.
- **Competency area:** Software/Systems/Application
- **Competency unit:** Design/Development/Improvement
- **Competency Statement:** Apply knowledge of programming paradigms, type systems, static and dynamic semantics, and the compilation/interpretation process to explain how a program executes as it should and does so in a safe and efficient manner.
- **Required knowledge areas and knowledge units:**
  - FPL/Object-Oriented Programming
  - FPL/Functional Programming
  - FPL/Logic Programming
  - FPL/Event-Driven and Reactive Programming
  - FPL/Advanced Programming Constructs
  - FPL/Type Systems
  - FPL Language Translation and Execution
  - FPL/Language Pragmatics
  - FPL/Embedded Computing and Hardware Interface
  - FPL/Parallel and Distributed Computing
  - FPL/Scripting
  - FPL/Language Translation and Execution
  - FPL/Program Representation
- **Required skill level:** Apply/Evaluate
- **Core level:**

- **Task 11:** Write a white paper explaining how a program executes in an efficient manner with respect to memory and CPU utilization.
- **Competency area:** Software/Systems/Application
- **Competency unit:** Design/Development/Improvement
- **Competency Statement:** Apply knowledge of programming paradigms, memory management, data representation and the compilation/interpretation process to explain how a program executes efficiently.
- **Required knowledge areas and knowledge units:**
  - FPL/Object-Oriented Programming
  - FPL/Functional Programming
  - FPL/Logic Programming
  - FPL/Event-Driven and Reactive Programming
  - FPL/Advanced Programming Constructs
  - FPL Language Translation and Execution
  - FPL/Language Pragmatics
  - FPL/Embedded Computing and Hardware Interface
  - FPL/Parallel and Distributed Computing
  - FPL/Scripting
  - FPL/Program Representation

- FPL/Program Representation
- **Required skill level:** Apply/Evaluate
- **Core level:**

Committee

Chair: Michael Oudshoorn, High Point University, NC, USA

Members:

- Annette Bieniusa, TU Kaiserslautern, Germany
- Brijesh Dongol, University of Surrey, UK
- Michelle Kuttel, University of Cape Town, South Africa
- Doug Lea, State University of New York at Oswego, NY, USA
- James Noble, Victoria University of Wellington, New Zealand
- Mark Marron, Microsoft Research, WA, USA
- Peter-Michael Osera, Grinnell College, IA, USA
- Michelle Mills Strout, University of Arizona, AZ, USA

Contributors:

- Alan Dearle, University of St. Andrews, Scotland

Appendix: Core Topics and Skill Levels

Knowledge Unit	Topic	Skill level	Core	Hours
Object-Oriented Programing	○ Object-oriented design • Decomposition into objects carrying state and having behavior • Class-hierarchy design for modeling ○ Definition of classes: fields, methods, and constructors ○ Subclasses, inheritance (including multiple inheritance), and method overriding ○ Dynamic dispatch: definition of method-call ○ Exception handling	Develop	CS	6

Knowledge Unit	Topic	Skill level	Core	Hours
	○ Object-oriented idioms for encapsulation • Privacy, data hiding, and visibility of class members • Interfaces revealing only method signatures • Abstract base classes, traits and mixins ○ Dynamic vs static properties ○ Composition vs inheritance			
	○ Subtyping • Subtype polymorphism; implicit upcasts in typed languages • Notion of behavioral replacement: subtypes acting like supertypes • Relationship between subtyping and inheritance ○ Collection classes, iterators, and other common library components	Develop	KA	3
Functional Programming	○ Lambda expressions and evaluation • Variable binding and scope rules • Parameter passing • Nested lambda expressions and reduction order ○ Effect-free programming • Function calls have no side effects, facilitating compositional reasoning • Immutable variables and data copying vs. reduction • Use of recursion vs. loops vs. pipelining (map/reduce) ○ Processing structured data (e.g., trees) via functions with cases for each data variant • Functions defined over compound data in terms of functions applied to the constituent pieces • Persistent data structures ○ Using higher-order functions (taking, returning, and storing functions)	Develop	CS	4

Knowledge Unit	Topic	Skill level	Core	Hours
	○ Function closures (functions using variables in the enclosing lexical environment) • Basic meaning and definition - creating closures at run-time by capturing the environment • Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments • Using a closure to encapsulate data in its environment • Lazy versus eager evaluation ○ Defining and implementing functions that can accept a function as a parameter, be combined with another function, or return a function.	Explain	KA	4
Logic Programming	• Universal vs. existential quantifiers • First order predicate logic vs. higher order logic • Expressing complex relations using logical connectives and simpler relations • Definitions of Horn clause, facts, goals, and subgoals • Unification and unification algorithm; unification vs. assertion vs expression evaluation • Mixing relations with functions • Cuts, backtracking and non-determinism • Closed-world vs. open-world assumptions	Explain	KA	3
Scripting	[*cross-reference: SDF and OS*] • Divide, combine, conquer • Fork • Error/exception handling • I/O redirection • System commands • Environment variables • File test operators • Data structures	Develop	CS	2

Knowledge Unit	Topic	Skill level	Core	Hours
	○ Arrays and lists ○ Slices ○ List Comprehensions • Regular expressions • Dynamic typing • Function declarations • Processes and threads • Code objects			
Event-driven and Reactive Programming	• Procedural programming vs. reactive programming: advantages of reactive programming in capturing events • Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers • Behavior model of event-based programming • Canonical uses such as GUIs, mobile devices, robots, servers • Reactive programs as state transition system	Develop	CS	2
	○ Using a reactive framework • Defining event handlers/listeners • Parameterization of event senders and event arguments • Externally-generated events and program-generated events ○ Separation of model, view, and controller	Develop	KA	2
Parallel and Distributed Computing	[*Cross-reference: PD/Parallel and Distributed Computing, SF/Parallelism*] • Safety and liveness ○ Race conditions ○ Dependencies/preconditions ○ Fault models ○ Termination • Parallel programming paradigms ○ Actor models ○ Task	Develop	CS	3

Knowledge Unit	Topic	Skill level	Core	Hours
	○ Message passing ○ Partitioned global address space ○ Procedural and reactive models ○ Synchronous/asynchronous programming absractions ○ Data parallelism • Semantics ○ Commutativity ○ Ordering ○ Independence ○ Consistency ○ Atomicity ○ Consensus • Execution control ○ Locks ○ Async await ○ Promises • Communication and coordination ○ Message-passing ○ Shared memory ○ cobegin-coend ○ Monitors ○ Channels ○ Threads ○ Guards			
	• Futures • Language support for data parallelism such as forall, , map/reduce • Loop unrolling • Effect of memory-consistency models on language semantics and correct code generation • Representational State Transfer Application Programming Interfaces (REST APIs) • Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing • Overheads of message passing	Explain	KA	2

Knowledge Unit	Topic	Skill level	Core	Hours
	• Granularity of program for efficient exploitation of concurrency. • Concurrency and other programming paradigms (e.g., functional).			

Type Systems	○ A type as a set of values together with a set of operations
	  • Primitive types (e.g., numbers, Booleans)
	  • Compound types built from other types (e.g., records, unions, arrays, lists, functions, references) using set operations
	○ Association of types to variables, arguments, results, and fields
	○ Type safety as an aspect of program correctness [*cross-reference: FPL/Formal Semantics*]
	○ Type safety and errors caused by using values inconsistently given their intended types
	○ Statically-typed vs dynamically-typed programming languages
	○ Type equivalence: structural vs name equivalence
	○ Goals and limitations of static and dynamic typing
	  • Detecting and eliminating errors as early as possible
	○ Generic types (parametric polymorphism) [*cross-reference: FPL/Formal Semantics*]
	  • Definition and advantages of polymorphism: parametric, subtyping, overloading and coercion
	  • Comparison of monomorphic and polymorphic types
	  • Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism
	  • Generic parameters and typing
	  • Use of generic libraries such as collections
		Develop	CS	3

Knowledge Unit	Topic	Skill level	Core	Hours
	• Comparison with ad hoc polymorphism (overloading) and subtype polymorphism • Prescriptive vs. descriptive polymorphism • Implementation models of polymorphic types			
	○ Complementary benefits of static and dynamic typing • Errors early vs. errors late/avoided • Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections • Typing rules ○ Rules for function, product, and sum types ○ Use of unification to find most general types ○ Lambda calculus plus let-polymorphism ○ Rules for function and type abstraction • Avoid misuse of code vs. allow more code reuse • Detect incomplete programs vs. allow incomplete programs to run • Relationship to static analysis • Decidability • Use of sophisticated (complex) type systems, e.g., Rust.	Develop	KA	4
Language Translation and Execution	○ Interpretation vs. compilation to native code vs. compilation to portable intermediate representation ○ Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution	Explain	CS	4

Knowledge Unit	Topic	Skill level	Core	Hours
	• BNF and extended BNF representation of context-free grammar • Parse tree using a simple sentence such as arithmetic expression or if-then-else statement • Ambiguity in Parsing due to lack of precedence order and resolution • Execution as native code or within a virtual machine • Alternatives like dynamic loading and dynamic (or "just-in-time") code generation o Control-flow diagrams using selection and iteration o Data structures for translation, execution, translation and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string o Direct, indirect and indexed access to memory location o Runtime representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects o Abstract low-level machine with simple instruction, stack and heap to explain translation and execution o Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap [*cross reference: AR/Memory System Architecture and Organization, OS/Memory Management*] • Translating selection and iterative constructs to control-flow diagrams • Translating control-flow diagrams to low level abstract code • Implementing loops, recursion, and tail calls • Translating function/procedure calls and return from calls, including different			

Knowledge Unit	Topic	Skill level	Core	Hours
	parameter passing mechanism using an abstract machine ○ Memory management [*cross reference: OS/Memory Management, FPL/Hardware Interface*] • Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects • Return from procedure as automatic deallocation mechanism for local data elements in the stack • Manual memory management: allocating, de-allocating, and reusing heap memory • Automated memory management: garbage collection as an automated technique using the notion of reachability			
Program Representation	• BNF and regular expressions • Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators • Components of a language ○ Definitions of alphabets, delimiters, sentences, syntax and semantics ○ Synatx vs. semantics • Types of semantics: operational, axiomatic, denotational, behavioral; define and use abstract syntax trees; contrast with concrete syntax • Program as a set of non-ambiguous meaningful sentences • Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling • Mutable vs. immutable variables: advantages and disadvantages of reusing existing	Explain	KA	3

Knowledge Unit	Topic	Skill level	Core	Hours
	memory location vs. advantages of copying and keeping old values; storing partial computation vs. recomputation • L-values and R-values: mapping mutable variable-name to L-values; mapping immutable variable-names to R-values • Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables • Scope rules: static vs. dynamic; visibility of variables; side-effects • Environment vs. store and their properties • Data and control abstraction • Mechanisms for information exchange between program units such as procedures, functions and modules: nonlocal variables, global variables, parameter passing, import-export between modules • Types of parameter passing with simple illustrations and comparison: call by value, call by reference, call by value-result, call by name, call by need and their variations • Side-effects induced by nonlocal variables, global variables and aliased variables • Data structures to represent code for execution, translation, or transmission • Low level instruction representation such as virtual machine instructions, assembly language, and binary representation [*cross-reference: AR/Machine Level Representation of Data*] • Lambda calculus, variable binding, and variable renaming. • String-based mobility in mobile coding.			

Graphics and Interactive Techniques (GIT)

Preamble

Computer graphics is the term used to describe the computer generation and manipulation of images and can be viewed as the science of enabling visual communication through computation. Its applications include machine learning; medical imaging; engineering; scientific, information, and knowledge visualization; cartoons; special effects; simulators; and video games. Traditionally, graphics at the undergraduate level focused on rendering, linear algebra, physics, the graphics pipeline, and phenomenological approaches. At the advanced level, undergraduate institutions are increasingly likely to offer one or more courses specializing in a specific graphics knowledge unit: e.g. gaming, animation, visualization, tangible or physical computing, and immersive courses such as AR/VR/XR. There is considerable overlap with other computer science knowledge areas: Artificial Intelligence; Human Computer Interaction; Parallel and Distributed Computing; Specialized Platform Development; and Society, Ethics and Professionalism.

In order for students to become adept at the use and generation of computer graphics, many implementation-specific issues must be addressed, such as human perception and cognition, data and image file formats, hardware interfaces, and application program interfaces (APIs). Undergraduate computer science students who study the knowledge units specified below through a balance of theory and applied instruction, will be able to understand, evaluate, and/or implement the related graphics and interactive techniques as users and developers. Because technology changes rapidly, the Graphics and Interactive Techniques subcommittee attempted to avoid being overly prescriptive. Where provided, examples of APIs, programs, and languages should be considered as appropriate examples in 2022. In effect, this is a snapshot in time.

Graphics as a knowledge area has expanded and become pervasive since the CS2013 report. Machine learning, computer vision, data science, AI, and the inclusion of embedded sensors in everything from cars to coffee makers utilize graphics and interactive techniques. The now ubiquitous cell phone has made the majority of the world's population regular users and creators of graphics, digital images, and immersive and interactive techniques. Animations, games, visualizations, and immersive applications that ran on desktops in 2013, now can run on mobile devices. The amount of data grew exponentially since 2013, and both data and visualizations are now published by myriad sources including news media and scientific organizations. Revenue from mobile video games now exceeds that of music and movies combined. Computer Generated Imagery (CGI) is employed in almost all films.[1]

[1] Jon Quast, Clay Bruning, and Sanmeet Deo. "Markets: This Opportunity for Investors Is Bigger Than Movies and Music Combined." retrieved from https://www.nasdaq.com/articles/this-opportunity-for-investors-is-bigger-than-movies-and-music-combined-2021-10-03.

It is critical that students and faculty confront the ethical questions and conundrums that have arisen and will continue to arise because of applications in computer graphics—especially those that employ machine learning, data science, and artificial intelligence. Today's news unfortunately provides examples of inequity and wrong-doing related to autonomous navigation, deep-space imaging, computational photography, deep fakes, and facial recognition.

Changes since CS 2013: In an effort to align CC2013's Graphics and Visualizations area with SIGGRAPH, we have renamed it Graphics and Interactive Techniques (GIT). To capture the expanding footprint of the field, the following knowledge units have been added to the original list of knowledge units (Fundamental Concepts, Visualization, Basic Rendering, Geometric Modeling, Advanced Rendering renamed Advanced Shading, Computer Animation):

- Immersion (MR, AR, VR)
- Interaction
- Image Processing
- Tangible/Physical Computing
- Simulation

Core Hours

Knowledge Unit	CS Core	KA Core
Fundamental Concepts	4	
Basic Rendering		18
Geometric Modeling		6
Advanced Shading		6
Computer Animation		Animation KA - 6
Visualization		Visualization KA - 6
Immersion (MR, AR, VR)		Immersion KA - 6
Interaction		Interaction KA - 6
Image Processing		Image Processing KA - 6
Tangible/Physical Computing		Tangible/Physical Computing KA - 6
Simulation		Simulation KA - 6
Total	**4**	

Knowledge Units

GIT/Fundamental Concepts

[4 CS Core hours]

For nearly every computer scientist and software developer, understanding of how humans interact with machines is essential. While these topics may be covered in a standard undergraduate graphics course, they may also be covered in introductory computer science and programming courses. Note that many of these topics are revisited in greater depth in later sections.

Topics:

[Core]

- Entertainment, business, and scientific applications: examples include visual effects, machine learning, computer vision, user interfaces, video editing, games and game engines, computer-aided design and manufacturing, data visualization, and virtual/augmented/mixed reality.
- Human vision system
    - tristimulus reception (RGB)
    - eye-as-camera (projection)
    - persistence of vision (frame rate/motion blur)
    - contrast (detection/Mach banding/dithering/aliasing)
    - non-linear response (tone mapping)
    - binocular vision (stereo)
    - accessibility (color deficiency, strobing, monocular vision etc.). (Cross-reference with SEP, HCI)
- Digitization of analog data
    - rasterization vs vector representations
        - example: polygon vs volume vs actual object
    - resolution
        - pixels for visual display
        - dots for laser printers
    - sampling and quantization
- Standard media formats
    - raster
        - lossless
        - lossy
    - vector
- Color models: additive (RGB), subtractive (CMYK) and color perception (HSV)
- Tradeoffs between storing data and re-computing data as embodied by vector and raster representations of images
- Animation as a sequence of still images
- SEP issues: deep fakes, facial recognition, privacy, intellectual property

Illustrative Learning Outcomes:

[CS Core]

1. Identify common uses of digital presentation to humans (e.g., computer graphics, sound).
2. Explain how analog signals can be reasonably represented by discrete samples, for example, how images can be represented by pixels.
3. Compute the memory requirement for storing a color image given its resolution.
4. Create a graphic depicting how the limits of human perception affect choices about the digital representation of analog signals.
5. Design a user interface and an alternative for persons with color perception deficiency.
6. Construct a simple user interface using a standard API.
7. When should you use each of the following common graphics file formats: JPG, PNG, MP3, MP4, and GIF? Why?
8. Give an example of a lossy and a lossless image compression technique found in common graphics file formats.
9. Describe color models and their use in graphics display devices.
10. Describe the tradeoffs between storing information vs. storing enough information to reproduce the information, as in the difference between vector and raster formats.
11. Compute the memory requirements for an *n* second movie based on *f* frames per second and *r* resolution.
12. Describe the basic process of producing continuous motion from a sequence of discrete frames (sometimes called “flicker fusion”).
13. Describe a possible visual mis-representation that could result from digitally sampling an analog world.

GIT/Visualization

Visualization is the process of creating graphical representations of data. Visualization has strong ties to Human Computer Interaction as well as Computational Science. Readers should refer to the HCI and CN KAs for additional topics related to user population and interface evaluations.

Topics:

- Data Visualization and Information Visualization
- Visualization of:
    - 2D/3D scalar fields
    - Vector fields and flow data
    - Time-varying data
    - High-dimensional data
    - Non-spatial data
- Visualization techniques (color mapping, isosurfaces, dimension reduction, parallel coordinates, multi-variate, tree/graph structured, text)
- Direct volume data rendering: ray-casting, transfer functions, segmentation.
- Common data formats (HDF, netCDF, geotiff, raw binary, CSV, ASCII to parse, etc.)
- Common Visualization software and libraries (R, Processing, D3.js, GIS, Matlab, IDL, Python, etc.)

- Perceptual and cognitive foundations that drive visual abstractions.
    - Visual communication
    - Color theory
- Visualization design.
    - Purpose (discovery, outreach).
    - Audience (technical, general public).
    - Ethically responsible visualization
        - Avoid misleading visualizations (exaggeration, hole filling, smoothing, data cleanup).
        - Even correct data can be misleading - eg, aliasing, incorrectly moving or stopped fan blades.
- Evaluation of visualization methods and applications.
- Visualization Bias
- Applications of visualization.

Illustrative Learning Outcomes:

1. Implement basic algorithms for visualization.
2. Evaluate the tradeoffs of visualization algorithms in terms of accuracy and performance.
3. Propose a suitable visualization design for a particular combination of data characteristics, application tasks, and audience.
4. Analyze the effectiveness of a given visualization for a particular task.
5. Design a process to evaluate the utility of a visualization algorithm or system.
6. Recognize a variety of applications of visualization including representations of scientific, medical, and mathematical data; flow visualization; and spatial analysis.

GIT/Basic Rendering

This section describes basic rendering and fundamental graphics techniques that nearly every undergraduate course in graphics will cover and that are essential for further study in most graphics-related courses.

Topics:

- Graphics pipeline.
- Rendering in nature, e.g., the emission and scattering of light and its relation to numerical integration.
- Forward and backward rendering (i.e., ray-casting and rasterization).
- Polygonal representation.
- Basic radiometry, similar triangles, and projection model.
- Affine and coordinate system transformations.
- Ray tracing.
- Visibility and occlusion, including solutions to this problem such as depth buffering, Painter’s algorithm, and ray tracing.
- The rendering equation.
- Simple triangle rasterization.
- Rendering with a shader-based API.

- Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping).
- Application of spatial data structures to rendering.
- Sampling and anti-aliasing.
- Scene graphs.

Illustrative Learning Outcomes:

1. Diagram the light transport problem and its relation to numerical integration i.e., light is emitted, scatters around the scene, and is measured by the eye.
2. Describe the basic rendering pipeline.
3. Compare and contrast how forward and backwards rendering factor into the graphics pipeline.
4. Create a program to display 2D shapes in a window.
5. Create a program to display 3D models.
6. Derive linear perspective from similar triangles by converting points (x, y, z) to points (x/z, y/z, 1).
7. Compute 2-dimensional and 3-dimensional points by applying affine transformations.
8. Apply the 3-dimensional coordinate system and the changes required to extend 2D transformation operations to handle transformations in 3D.
9. Explain the concept and applications of texture mapping, sampling, and anti-aliasing.
10. Compare ray tracing and rasterization for the visibility problem.
11. Implement simple procedures that perform transformation and clipping operations on simple 2-dimensional images.
12. Implement a simple real-time renderer using a rasterization API (e.g., OpenGL) using vertex buffers and shaders.
13. Compare and contrast the different rendering techniques.
14. Compute space requirements based on resolution and color coding.
15. Compute time requirements based on refresh rates and rasterization techniques.

GIT/Geometric Modeling

Geometric modeling includes the representation, creation and manipulation of 2D shapes and 3D forms.

Topics:

- Basic geometric operations such as intersection calculation and proximity tests
- Surface representation/model
    - Tessellation
    - Mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes
    - Parametric polynomial curves and surfaces
    - Implicit representation of curves and surfaces
    - Spatial subdivision techniques
- Volumetric representation/model
    - Volumes, voxels, and point-based representations.
    - Signed Distance Fields

    - Sparse Volumes, i.e., VDB
    - Constructive Solid Geometry (CSG) representation
- Procedural representation/model
    - Fractals
    - L-Systems, cross referenced with programming languages (grammars to generated pictures).
- Procedural models such as fractals, generative modeling
- Elastically deformation and freeform deformable models.
    - Quasi-static methods
    - Bi-harmonic capture/deform
- Multiresolution modeling.
- Reconstruction.

Illustrative Learning Outcomes:

1. Contrast representing curves and surfaces in both implicit and parametric forms.
2. Create simple polyhedral models by surface tessellation.
3. Generate a mesh representation from an implicit surface.
4. Generate a fractal model or terrain using a procedural method.
5. Generate a mesh from data points acquired with a laser scanner.
6. Construct CSG models from simple primitives, such as cubes and quadric surfaces.
7. Contrast modeling approaches with respect to space and time complexity and quality of image.

GIT/Shading

Topics:

- Solutions and approximations to the rendering equation, for example:
    - Distribution ray tracing and path tracing
    - Photon mapping
    - Bidirectional path tracing
    - Metropolis light transport
- Time (motion blur), lens position (focus), and continuous frequency (color) and their impact on rendering
- Shadow mapping
- Occlusion culling
- Bidirectional Scattering Distribution function (BSDF) theory and microfacets
- Subsurface scattering
- Area light sources
- Hierarchical depth buffering
- The Light Field, image-based rendering
- Non-photorealistic rendering
- GPU architecture
- Human visual systems including adaptation to light, sensitivity to noise, and flicker fusion

Learning Outcomes:

1. Demonstrate how an algorithm estimates a solution to the rendering equation.
2. Prove the properties of a rendering algorithm, e.g., complete, consistent, and unbiased.
3. Analyze the bandwidth and computation demands of a simple algorithm.
4. Implement a non-trivial shading algorithm (e.g., toon shading, cascaded shadow maps) under a rasterization API.
5. Show how a particular artistic technique might be implemented in a renderer.
6. Explain how to recognize the graphics techniques used to create a particular image.
7. Implement any of the specified graphics techniques using a primitive graphics system at the individual pixel level.
8. Implement a ray tracer for scenes using a simple (e.g., Phong's) BRDF plus reflection and refraction.

GIT/Computer Animation

Topics:

- Principles of Animation (Squash and Stretch, Timing, Anticipation, Staging, Follow Through and Overlapping Action, Straight Ahead Action and Pose-to-Pose Action, Slow In and Out, Arcs, Exaggeration, and Appeal)
- Key-frame animation
    - Keyframe Interpolation Methods: Lerp / Slerp / Spline
- Forward and inverse kinematics
- Skinning algorithms
    - Capturing
    - Linear blend, dual quaternion
- Rigging
- Blend shapes
    - Pose space deformation
- Transforms:
    - Translations
    - Scale / Shear
    - Rotations
        - Euler angles
        - Quaternions
        - Angle/axis, exponential map
    - Transformation Order: SRT / XYZ
- Camera animation
    - Look at
    - Focus
- Motion capture
    - Set up and fundamentals
    - Ethical considerations (e.g., accessibility and privacy)
        - Avoidance of "default" captures - there is no typical human walk cycle.

Learning Outcomes:

1. Compute the location and orientation of model parts using a forward kinematic approach.
2. Compute the orientation of articulated parts of a model from a location and orientation using an inverse kinematic approach.
3. Compare the tradeoffs in different representations of rotations.
4. Implement the spline interpolation method for producing in-between positions and orientations.
5. Use common animation software to construct simple organic forms using metaball and skeleton.

GIT/Simulation

Simulation has strong ties to Computational Science. In the graphic domain, however, simulation techniques are re-purposed to a different end. Rather than creating predictive models, the goal instead is to achieve a mixture of physical plausibility and artistic intention. The goals of “model surface tension in a liquid” and “produce a crown splash” are related, but different.

Topics by Subject:

- Collision detection and response
    - Signed Distance Fields
    - Sphere/sphere
    - Triangle/point
    - Edge/edge
- Procedural animation using noise
- Particle systems
    - Integration methods (Forward Euler, Midpoint, Leapfrog)
    - Mass/spring networks
    - Position based dynamics
    - Rules (boids/crowds)
    - Rigid bodies
- Grid based fluids
    - Semi-Lagrangian advection
    - Pressure Projection
- Heightfields
    - Terrain: Transport, erosion
    - Water: Ripple, Shallow water.
- Rule-based system
    - LSystems.
    - Space-colonizing systems.
    - Game of Life

Prerequisite Data Structures:

- Dense Volumes
- Sparse Volumes
- Adaptive Volumes

- Points with Attributes
- Triangle Soups
- Heightfields

Goals (Given a goal, which topics should be used):

- Particle systems
    - Integration methods (Forward Euler, Midpoint, Leapfrog)
- Rigid Body Dynamics
    - Particle systems
    - Collision Detection
        - Tri/point, edge/edge
- Cloth
    - Particle systems
    - Mass/spring networks
    - Collision Detection
        - Tri/point, edge/edge
- Particle-Based Water
    - Integration methods
    - Smoother Particle Hydrodynamics (SPH) Kernels
    - Signed Distance Function-Based Collisions
- Grid-Based Smoke and Fire
    - Semi-Lagrangian Advection
    - Pressure Projection
- Grid and Particle-Based Water
    - Particle-Based Water
    - Grid-Based Smoke, and Fire

Illustrative Learning Outcomes:

1. Implement algorithms for physical modeling of particle dynamics using simple Newtonian mechanics, for example Witkin & Kass, snakes and worms, symplectic Euler, Stormer/Verlet, or midpoint Euler methods.
2. Contrast the basic ideas behind fluid simulation methods for modeling ballistic trajectories, for example for splashes, dust, fire, or smoke.
3. Implement a smoke solver with user interaction

GIT/Immersion

Topics:

- Define and distinguish VR, AR, and MR
- Stereoscopic display
- Viewer tracking
    - Inside out vs Outside In
    - Head / Body / Hand / tracking
- Visibility computation
- Time-critical rendering, multiple levels of details (LOD) Image-based VR system

    - Motion to Photon latency
- Distributed VR, collaboration over computer network
- Interactive modeling
- Applications in medicine, simulation, training, and visualization
- Safety in immersive applications
    - Motion sickness
    - VR obscures the real world, which increases the potential for falls and physical accidents
- Accessibility in immersive applications
    - Accessible to those who cannot move
    - Accessible to those who cannot be moved
- Ethics/privacy in immersive applications. (cross-reference with SEP)
    - Acquisition of private data (room scans, body proportions, active cameras, etc)
    - Can't look away from immersive applications easily.
    - Danger to self/surroundings while immersed

Illustrative Learning Outcomes:

1. Create a stereoscopic image.
2. Summarize the pros and cons of different types of viewer tracking.
3. Compare and contrast the differences between geometry- and image-based virtual reality.
4. Judge and defend the design issues of user action synchronization and data consistency in a networked environment.
5. Create the specifications for an augmented reality application to be used by surgeons in the operating room.
6. Evaluate an immersive application's accessibility (cross-reference with HCI)
7. Identify the most important technical characteristics of a VR system/application that should be controlled to avoid motion sickness and explain why.

GIT/Interaction

Interactive computer graphics is a requisite part of real time applications ranging from the utilitarian like word processors to virtual and/or augmented reality applications.
Students will learn the following topics in a graphics course or a course that covers HCI/GUI Construction and HCI/Programming.

Topics:

- Event Driven Programming
    - Mouse or touch events
    - Keyboard events
    - Voice input
    - Sensors
    - Message passing communication
    - Network events
    - Interrupt event processing
- Graphical User Interface (Single Channel)

    - Window
    - Icons
    - Menus
    - Pointing Devices
- Gestural Interfaces
    - Accessibility - other approaches if gesture not possible (Inject “thumbs up” without a thumb)
- Haptic Interfaces
    - External actuators
    - Gloves
    - Exoskeletons
- Multimodal Interfaces
- Immersive Interfaces (AI)
    - brainwave (EEG type electrodes)
    - headsets with embedded eye tracking
    - AR glasses
- Accessibility (cross-reference with SEP)

Illustrative Learning Outcomes:

- Create a simple game that responds to single channel mouse and keyboard events
- Program a circuit to respond to a variable resistor
- Create a mobile app that responds to touch events
- Use gestures to control a program
- Design and implement an application that provides haptic feedback
- Design and implement an application that responds to different event triggers

GIT/Image Processing

Image Processing consists of the analysis and processing of images for multiple purposes, but most frequently to improve image quality and to manipulate imagery. It is the cornerstone of Computer Vision which is a KU in the AI.

Topics:

- Morphological operations
    - Connected components
    - Dilation
    - Erosion
    - Computing region properties (area, perimeter, centroid, etc.)
- Color histograms
    - Representation
    - Contrast enhancement through normalization
- Image enhancement
    - Convolution
    - Blur (e.g., Gaussian)
    - Sharpen (Laplacian)
    - Frequency filtering (low-pass, high-pass)

- Image restoration
    - Noise, degradation
    - Inpainting and other completion algorithms
    - Wiener filter
- Image coding
    - Redundancy
    - Huffman coding
    - DCT, wavelet transform, Fourier transforms
    - Nyquist Theorem
    - Watermarks
        - Ethical considerations
- Emerging area:
    - Convolutional Neural Networks
    - Transformers
- SEP issues
    - Deep fakes
    - Applications that misidentify people based on skin color or hairstyle

Illustrative Learning Outcomes:

- Use dilation and erosion to smooth the edges of a binary image.
- Manipulate hue in an image
- Filter an image using a high-pass filter (advanced: in frequency domain)
- Restore missing part of an image using an inpaint algorithm (e.g., Poisson image editing)
- Enhance an image by selectively filtering in the frequency domain
- Describe the ethical pitfalls of facial recognition. Can facial recognition be used ethically? If so, how?

GIT/Tangible/Physical Computing

Cross-cutting with Specialized Platform Development and HCI

Topics:

- Communication with the physical world
    - Acquisition of data from sensors
    - Driving external actuators
- Event driven programming (see Interaction topic)
- Connection to physical artifacts
    - Computer Aided Design
    - Computer Aided Manufacturing
    - Fabrication
        - HCI prototyping (see HCI)
        - Additive (3D printing)
        - Subtractive (CNC milling)
        - Forming (vacuum forming)
- Internet of Things (reference Networking)

    - Network connectivity
    - Wireless communication
- SEP issue
    - Privacy

Illustrative Learning Outcomes:

- Construct a simple switch and use it to turn on an LED.
- Construct a simple system to move a servo in response to sensor data
- Use a light sensor to vary a property of something else (e.g. color or brightness of an LED or graphic)
- Create a 3D form in a CAD package
    - Show how affine transformations are achieved in the CAD program
    - Show an example of instances of an object
    - Create a fabrication plan. Provide a cost estimate for materials and time. How will you fabricate it?
    - Fabricate it. How closely did your actual fabrication process match your plan? Where did it differ?.
- Write the G- and M-Code to construct a 3D maze, use a CAD/CAM package to check your work
- If you were to design an IoT pill dispenser, would you use Ethernet, WiFi, Bluetooth. RFID/NFC, or something else for Internet connectivity. Why? Make one.
- Distinguish between the different types of fabrication and describe when you would use each.

Professional Dispositions

- Proactive: take initiative, self-starter, independent
- Self-directed: self-learner, self-motivated
- Goal-driven
- Professional: exercise discretion, behave ethically
- Adaptable: Flexible
- Collaborative: team player
- Responsive
- Meticulous: attentive to detail, thorough
- Inventive: look beyond simple solutions
- Accountable for decisions and their ramifications
- The ability to give and receive code reviews
- Effective communication
    - oral
    - written
    - code

Math Requirements

Required:

- Linear Algebra:
  - Points (coordinate systems & homogeneous coordinates), vectors, and matrices
  - Vector operations: addition, scaling, dot and cross products
  - Matrix operations: addition, multiplication, determinants
  - Affine transformations
- Calculus
  - Continuity

Desirable:

- Linear Algebra
  - Eigenvectors and and Eigen decomposition
  - Gaussian Elimination and Lower Upper Factorization
  - Singular Value Decomposition
- Calculus
  - Quaternions

Necessary and Desirable Data Structures

Data Structures necessary for this knowledge area includes:

- Directed Acyclic Graphs
- Tuples (Points / vectors / matrices of fixed dimension)
- Dense 1d, 2d, 3d arrays.

Data Structures desirable for this knowledge area includes:

- Array of Structures vs Structure of Arrays
- Trees (e.g. K-trees, quadtrees, Huffman Trees)

Shared Concepts and Crosscutting Themes

Shared Concepts:

- GIT Immersion and HCI
- GIT Interaction and HCI/GUI Programming and CN/Interactive Visualization
- Graftals (GIT/Modeling) with Programming Languages (PL/BNF grammars)
- Simulation
- Visualization in GIT, AI, and Specialized Platform Development Interactive Computing Platforms (Data Visualization)
- Image Processing in GIT and Specialized Platform Development Interactive Computing Platforms (Supporting Math Studies)
- Tangible Computing in GIT and Specialized Platform Development Interactive Computing Platforms (Game Platforms)

- Tangible Computing in GIT and Specialized Platform Development Interactive Computing Platforms (Embedded Platforms)
- Tangible Computing and Animation in GIT and Specialized Platform Development Interactive Computing Platforms (Robot Platforms)
- Immersion in GIT and Specialized Platform Development Interactive Computing Platforms (Mobile Platforms)
- Image Processing in GIT and Advanced Machine Learning (Graphical Models) in AI
- Image Processing and Physical Computing in GIT and Robotics Location and Mapping and Navigation in AI
- Image Processing in GIT and Perception and Computer Vision in AI
- Core Graphics in GIT and Algorithms and Application Domains in PD
- GIT and Interactive Computing Platforms in SPD
- GIT and Game Platforms in SPD
- GIT and Imbedded Platforms in SPD

Crosscutting themes:

- Efficiency
- Ethics
- Modeling
- Programming
- Prototyping
- Usability
- Evaluation

Competency Specifications

- **Task 1:** Select the correct image format.
- **Competency Statement:** Select the image format most appropriate for 1) a photograph, 2) a cartoon, and 3) a cut path for a water jet.
- **Competency area:** Application / Theory
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Core
- **Required skill level:** Understand and Explain
- **Core level:**

- **Task 2:** Write a white paper describing the rendering pipeline.
- **Competency Statement:** In general terms explain what happens between the graphics API and the frame buffer.
- **Competency area:** Application
- **Competency unit:** Communicate
- **Required knowledge areas and knowledge units:**

    - KA-GIT / KU Fundamental Concepts
    - KA-GIT / KU Basic Rendering
- **Required skill level:** Understand and Explain
- **Core level:**

- **Task 3:** Animate a bouncing ball.
- **Competency Statement:** Animate a realistically bouncing kickball. Pay attention to the principles of animation.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Fundamental Concepts
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU Animation
    - KA-GIT / KU Advanced Shading
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 4:** Create an organic-looking striped pattern.
- **Competency Statement:** Using your favorite programming language, write a program to implement a reaction diffusion model to produce stripes. Design the user interface so that the parameters can be manipulated in real time.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Core
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU Advanced Shading
    - KA-GIT / KU Simulation
    - KA-GIT / KU Interaction
    - KA-HCI / KU-System Design
    - KA-HCI / KU-Understanding the User
    - KA-SDF / KU-Development Methods
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 5:** Create a digital fireworks display customizable by the user.
- **Competency Statement:** Using your favorite programming language, write a program to simulate fireworks using a particle system. Design the user interface so that the majority of parameters are adjustable in real time.
- **Competency area:** Application

- **Competency unit:** Requirements / Design / Development / Testing / Deployment
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Core
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU Simulation
    - KA-GIT / KU Interaction
    - KA-HCI / KU-System Design
    - KA-HCI / KU-Understanding the User
    - KA-SDF / KU-Development Methods
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 6:** Simulate impaired vision.
- **Competency Statement:** Use Gaussian filtering of varying kernel sizes to simulate near-sightedness.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Consumer Acceptance
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Core
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU Image Processing
    - KA-SDF / KU-Development Methods
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 7:** Enhance an image lacking contrast.
- **Competency Statement:** Use histogram normalization on a three-channel color image.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Fundamental Concepts
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU Image Processing
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 8:** Create a tic-tac-toe app.
- **Competency Statement:** Create a tic-tac-toe mobile app for both iphone and Android platforms designed for people 6 years of age and older.
- **Competency area:** Application

- **Competency unit:** Requirements / Design / Development / Testing / Deployment
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Fundamental Concepts
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU Interaction
    - KA-HCI / KU-System Design
    - KA-HCI / KU-Understanding the User
    - KA-SPD / Mobile Foundations
    - KA-SPD / Mobile Platforms
    - KA-SDF / KU-Development Methods
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

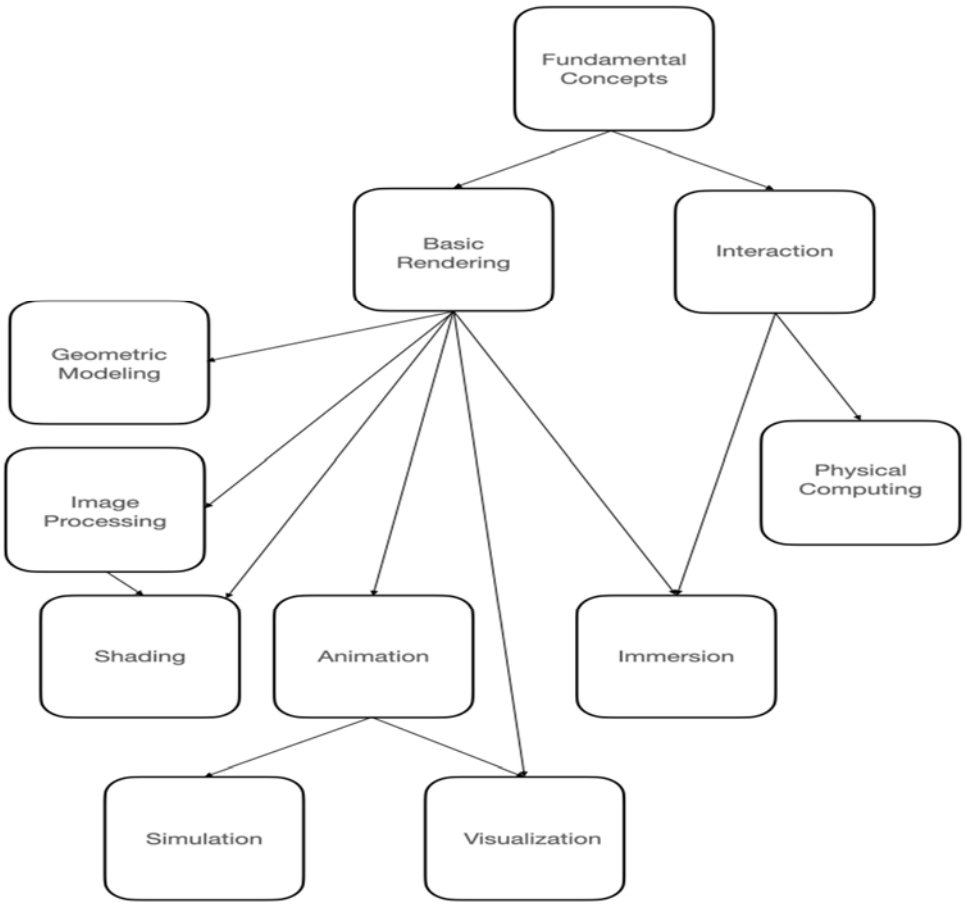
- **Task 9:** Visualize a region's temperature
- **Competency Statement:** Given weather data, design and implement an animation depicting temperature changes for a region over time.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Evaluation / Consumer Acceptance
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Fundamental Concepts
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU-Visualization
    - KA-HCI / KU-System Design
    - KA-HCI / KU-Understanding the User
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 10:** Visually compare the COVID 19 infections over time in two locations.
- **Competency Statement:** Given COVID 19 data, design and implement an animation depicting the number of infections in two locations over time so that they can be compared.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Evaluation / Consumer Acceptance
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU Fundamental Concepts
    - KA-GIT / KU Basic Rendering
    - KA-GIT / KU-Visualization
    - KA-HCI / KU-System Design
    - KA-HCI / KU-Understanding the User
- **Required skill level:** Apply / Evaluate / Develop

- **Core level:**

- **Task 11:** Create a device to turn on an LED in low light.
- **Competency Statement:** Using a microcontroller and standard electronic components, create an electrical circuit and program the microcontroller to turn on an LED in low light.
- **Competency area:** Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment
- **Required knowledge areas and knowledge units:**
    - KA-GIT / KU-Fundamental Concepts
    - KA-GIT / KU-Physical Computing
    - KA-SPD / KU-Embedded Platforms
    - KA-HCI / KU-System Design
    - KA-SEP, KU-Privacy
- **Required skill level:** Explain / Apply / Evaluate / Develop
- **Core level:**

Course Packaging Suggestions



Interactive Computer Graphics to include the following:

- GIT KU Basic Rendering: 40 hours
- SEP KUs Ethics: 4 hours

Pre-requisites:

- CS2
- Affine Transforms from Linear Algebra
- Trigonometry

Skill statement: A student who completes this course should understand and be able to create basic computer graphics using an API. They should know how to position and orient models, the camera, and distant and local lights.

Tangible Computing to include the following:

- GIT KU Physical Computing: 30 hours
- SPD KU Embedded Platforms: 10 hours
- SEP KU Privacy and DEI: 4 hours

Pre-requisites:

- CS1

Skill statement: A student who completes this course should be able to design and build circuits and program a microcontroller. They will understand polarity, Ohm's law, and how to work with electronics safely.

Image Processing to include the following:

- GIT KU Image Processing: 30 hours
- GIT KU Basic Rendering: 10 hours
- KUs DEI, Privacy, Intellectual Property Law from SEP Knowledge Area: 4 hours

Pre-requisites:

- CS2
- Linear Algebra

Skill statement: A student who completes this course should understand and be able to appropriately acquire, process, display, and save digital images.

Data Visualization to include the following:

- GIT KU Visualization: 30 hours
- GIT KU Basic Rendering: 10 hours
- HCI, KU Understanding the User: 3 hours
- KUs DEI, Privacy, Ethics from SEP Knowledge Area: 4 hours

Pre-requisites:

- CS2
- Linear Algebra

Skill statement: A student who completes this course should understand how to select a dataset; ensure the data are accurate and appropriate; design, develop and test a visualization program that depicts the data and is usable.

Simulation to include the following:

- GIT KU Simulation: 30 hours
- GIT KU Basic Rendering: 10 hours
- SEP KUs Ethics: 4 hours

Pre-requisites:

- CS2
- Linear Algebra

Skill statement: A student who completes this course should understand and be able to create directable simulations, both of physical and non-physical systems.

Course Packaging Suggestions

Chair: Susan Reiser, UNC Asheville, Asheville, USA

Members:

- Erik Brunvand, University of Utah, Salt Lake City, USA
- Kel Elkins, NASA/GSFC Scientific Visualization Studio, Greenbelt, MD
- Jeff Lait, SideFX, Toronto, Canada

- Amruth Kumar, Ramapo College, Mahwah, USA
- Paul Mihail, Valdosta State University, Valdosta, USA
- Tabitha Peck, Davidson College, Davidson, USA
- Ken Schmidt, NOAA NCEI, Asheville, USA
- Dave Shreiner, Unity, San Francisco, USA

Contributors:

- Greg Shirah, NASA/GSFC Scientific Visualization Studio, Greenbelt, MD
- AJ Christensen, NASA/GSFC Scientific Visualization Studio – SSAI, Champaign, IL
- Barbara Mones, University of Washington, Seattle, WA, USA
- Beatriz Sousa Santos, University of Aveiro, Portugal
- Ted Kim, Yale University, CT, USA
- Ginger Alford, Southern Methodist University, TX, USA

Appendix: Core Topics and Skill Levels

KU	Topic	Skill	Core	Hours
Core	• Applications • Human vision system • Digitization of analog data • Standard media formats • Color Models • Tradeoffs between storing data and re-computing data • Animation as a sequence of still images • SEP related to graphics	Explain	CS	4

Basic Rendering	● Graphics pipeline. ● Affine and coordinate system transformations. ● Rendering in nature, e.g., the emission and scattering of light and its relation to numerical integration. ● Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping). ● Sampling and anti-aliasing. ● Visibility and occlusion, including solutions to this problem such as depth buffering, Painter's algorithm, and ray tracing.	Explain	KA	10
	● Forward and backward rendering (i.e., ray-casting and rasterization). ● Polygonal representation. ● Basic radiometry, similar triangles, and projection model. ● Ray tracing. ● The rendering equation. ● Simple triangle rasterization. ● Application of spatial data structures to rendering. ● Scene graphs.	Explain	KA	5
	● Generate an image with a standard API	Implement	KA	3
Visualization KA Core	● Visualization of: ○ 2D/3D scalar fields: color mapping ○ Time-varying data	Explain and Implement	KA	3

	• Visualization techniques (color mapping, dimension reduction)	Explain and Implement	KA	2
	• Perceptual and cognitive foundations that drive visual abstractions.	Explain	KA	1
	• Visualization Bias	Evaluate	KA	1

Modeling KA Core	• Surface representation/model
	  • Mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes
	  • Parametric polynomial curves and surfaces
		Explain and Use	KA	2

	• Volumetric representation/model
	  • Volumes, voxels, and point-based representations.
	  • Constructive Solid Geometry (CSG) representation
		Explain and Use	KA	2

	• Procedural representation/model
	  • Fractals
	  • L-Systems, cross referenced with programming languages (grammars to generated pictures).
	  • Generative Modeling
		Explain and Use	KA	2

Shading KA Core	• Time (motion blur) and lens position (focus) and their impact on rendering • Shadow mapping • Occlusion culling • Area light sources • Hierarchical depth buffering • Non-photorealistic rendering	Explain and Use	KA	6

Computer Animation KA Core	• Principles of Animation (Squash and Stretch, Timing, Anticipation, Staging, Follow Through and Overlapping Action, Straight Ahead Action and Pose-to-Pose Action, Slow In and Out, Arcs, Exaggeration, and Appeal) • Key-frame animation	Explain and Use	KA	2
	• Forward and inverse kinematics	Explain and Use	KA	2

	• Transforms:
	  • Translations
	  • Scale / Shear
	  • Rotations
	• Camera animation
	  • Look at
	  • Focus
		Explain and Implement	KA	2

Simulation KA Core	• Particle systems	Explain and implement	KA	2
	• Collision detection and response	Explain and Implement	KA	2
	• Grid based fluids	Explain and Implement	KA	2
Immersion KA Core	• Define and distinguish VR, AR, and MR • Applications in medicine, simulation, and training, and visualization	Explain	KA	1

	• Stereoscopic display
	• Viewer tracking
	  ○ Inside out vs Outside In
	  ○ Head / Body / Hand / tracking
	• Visibility computation
		Explain and Implement	KA	3

	• Safety in immersive applications • Accessibility in immersive applications. • Ethics/privacy in immersive applications.	Explain and Evaluate	KA	2
Interaction KA Core	• Event Driven Programming (Shared with SPD Interactive, SPD Game Development) • Graphical User Interface	Explain and Implement	KA	4
	• Accessibility in GUI Design (shared with HCI)	Explain and Evaluate	KA	2
Image Processing	• Convolution filters	Explain and implement	KA	2
	• Convolutional Neural Networks.(shared with AI:Machine Learning)	Explain and Implement	KA	2
	• Histograms • Fourier and/or Cosine Transforms	Explain and Implement	KA	2

Physical Computing KA Core	• Communication with the physical world (shared with SPD/Embedded Systems, PL/Embedded Systems)
	  ○ Acquisition of data from sensors
	  ○ Driving external actuators
		Explain and Implement	KA	1

	• Event driven programming (shared with GIT: Interaction)	Explain and Implement	KA	1

	• Connection to physical artifacts
	  ○ Computer Aided Design
	  ○ Computer Aided Manufacturing
	  ○ Fabrication
	  ■ prototyping (shared with HCI)
	  ■ Additive (3D printing)
	  ■ Subtractive (CNC milling)
	  ■ Forming (vacuum forming)
		Explain, evaluate, and Implement an example	KA	3

	● Internet of Things ○ Network connectivity ○ Wireless communication	Explain	KA	1

Human-Computer Interaction (HCI)

Preamble

Computational systems not only enable users to solve problems, but also foster social connectedness and support a broad variety of human endeavors. Thus, these systems should interact with their users and solve problems in ways that respect individual dignity, social justice, and human values and creativity. Human-computer interaction (HCI) addresses those issues from an interdisciplinary perspective that includes psychology, business strategy, and design principles.

Each user is different and, from the perspective of HCI, the design of every system that interacts with people should anticipate and respect that diversity. This includes not only accessibility, but also cultural and societal norms, neural diversity, modality, and the responses the system elicits in its users. An effective computational system should evoke trust while it treats its users fairly, respects their privacy, provides security, and abides by ethical principles.

These goals require design-centric engineering that begins with intention and with the understanding that design is an iterative process, one that requires repeated evaluation of its usability and its impact on its users. Moreover, technology evokes user responses, not only by its output, but also by the modalities with which it senses and communicates. This knowledge area heightens the awareness of these issues and should influence every computer scientist.

Changes since CS 2013: Driven by this broadened perspective, the HCI knowledge area has revised the CS 2013 document in several ways:

- Knowledge units have been renamed and reformulated to reflect current practice and to anticipate future technological development.
- There is increased emphasis on the nature of diversity and the centrality of design focused on the user.
- Modality (e.g., text, speech) is still emphasized given its key role throughout HCI, but with a reduced emphasis on particular modalities in favor of a more timely and empathetic approach.
- The curriculum reflects the importance of understanding and evaluating the impacts and implications of a computational system on its users, including issues in ethics, fairness, trust, and explainability.
- Given its extensive interconnections with other knowledge areas, we believe HCI is itself a cross-cutting knowledge area with connections to Artificial Intelligence; Society, Ethics and Professionalism, Software Development Fundamentals, Software Engineering.

Core Hours

Knowledge Unit	CS Core	KA Core
Understanding the User	2	5
Accountability and Responsibility in Design	2	2
Accessibility and Inclusive Design	2	2
Evaluating the Design	1	2
System Design	1	5
Total Hours	**8**	**16**

Knowledge Units

Knowledge Unit 1: Understanding the User: Individual goals and interactions with others

- CS Core Topics
  - **User-centered design and evaluation methods:** "you are not the users"**,** user needs-finding, formative studies, interviews, surveys, usability tests
- KA Core Topics
  - **User-centered design methodology:** personas/persona spectrum; user stories/storytelling and techniques for gathering stories; empathy maps; needs assessment (techniques for uncovering needs and gathering requirements - e.g., interviews, surveys, ethnographic and contextual enquiry); journey maps. See also: Knowledge Unit on Evaluating the Design.
  - **Physical & cognitive characteristics of the user:** physical capabilities that inform interaction design (e.g., color perception, ergonomics); cognitive models that inform interaction design (e.g., attention, perception and recognition, movement, memory); topics in social/behavioral psychology (e.g., cognitive biases, change blindness).

    - **Designing for diverse user populations:** how differences (e.g., in race, ability, age, gender, culture, experience, and education) impact user experiences and needs; internationalization, designing for users from other cultures, and cross-cultural design; challenges to effective design evaluation (e.g., sampling, generalization; disability and disabled experiences); universal design. See also: Knowledge Unit on Accessibility and Inclusive Design.
    - **Collaboration and communication:** understanding the user in a multi-user context; synchronous group communication (e.g., chat rooms, conferencing, online games); asynchronous group communication (e.g., email, forums, social networks); social media, social computing, and social network analysis; online collaboration, social coordination, and online communities; avatars, characters, and virtual worlds.
- Non-core Topics (including Emerging topics)

Illustrative Learning Outcomes

- CS Core
    - Conduct a user-centered design process that is integrated into a project.
- KA Core
    - Compare and contrast the needs of users with those of designers.
    - Identify the representative users of a design and discuss who else could be impacted by it.
    - Describe empathy and evaluation as elements of the design process.
    - Carry out and document an analysis of users and their needs.
    - Construct a user story from a needs assessment.
    - Redesign an existing solution to a population whose needs differ from those of the initial target population.
    - Contrast the different needs-finding methods for a given design problem.
    - Reflect on whether your design would benefit from low-tech or no-tech components.
- Non-core
    - Recognize the implications of designing for a multi-user system/context.

Knowledge Unit 2: Accountability and Responsibility in Design: Sustainability, security, privacy, trust, and ethics

- CS Core Topics
    - **Design impact**: sustainability, inclusivity, safety, security, privacy, harm, and disparate impact.
    - **Ethics:** in design methods and solutions; the role of artificial intelligence; responsibilities for considering stakeholder impact and human factors, role of design to meet user needs.

    - **Requirements in design:** ownership responsibility, legal frameworks, compliance requirements, consideration beyond immediate user needs, including via iterative reconstruction of problem analysis..
- KA Core Topics
    - **Value-sensitive design:** identify direct and indirect stakeholders, determine and include diverse stakeholder values and value systems.
    - **Persuasion through design**: assessing the persuasive content of a design, persuasion as a design goal.
- Non-core Topics (including Emerging topics)

Illustrative Learning Outcomes

- CS Core
    - Identify and critique the potential impacts of a design on society and relevant communities to address such concerns as sustainability, inclusivity, safety, security, privacy, harm, and disparate impact
- KA Core
    - Identify the potential human factor elements in a design.
    - Identify and understand direct and indirect stakeholders.
    - Develop scenarios that consider the entire lifespan of a design, beyond the immediately planned uses that anticipate direct and indirect stakeholders.
    - Identify and critique the potential factors in a design that impact direct and indirect stakeholders and broader society (e.g., transparency, sustainability of the system, trust, artificial intelligence).
    - Assess the persuasive content of a design and its intent relative to user interests.
    - Critique the outcomes of a design given its intent.
    - Understand the impact of design decisions
- Non-core

Knowledge Unit 3: Accessibility and Inclusive Design

- CS Core Topics
    - **Background:** societal and legal support for and obligations to people with disabilities; accessible design benefits everyone.
    - **Techniques**: accessibility standards such as Web Content Accessibility Guidelines.
    - **Technologies**: features and products that enable accessibility and support inclusive development by designers and engineers
    - **Inclusive Design Frameworks:** recognizing differences; universal design
- KA Core Topics

    - **Background:** demographics and populations (permanent, temporary and situational disability); international perspectives on disability; attitudes towards people with disabilities
    - **Techniques**: UX (user experience) design and research; software engineering practices that enable inclusion and accessibility.
    - **Technologies**: examples of accessibility-enabling features, such as conformance to screen readers
    - **Inclusive Design Frameworks:** creating inclusive processes such as participatory design; designing for larger impact.
- Non-core Topics (including Emerging topics)
    - **Background:** unlearning and questioning; disability studies
    - **Technologies**: the Return on Investment of inclusion
    - **Inclusive Design Frameworks:** user-sensitive inclusive design
    - **Critical approaches to HCI:** critical race theory in HCI, feminist HCI, critical disability theory.

Illustrative Learning Outcomes

- CS Core
    - Identify accessibility challenges faced by people with different disabilities, and specify the associated accessible and assistive technologies that address them.
    - Identify appropriate inclusive design approaches, such as universal design and ability-based design.
    - Identify and demonstrate understanding of software accessibility guidelines.
    - Demonstrate recognition of laws and regulations applicable to accessible design.
- KA Core
    - Apply inclusive frameworks to design, such as universal design and usability and ability-based design, and demonstrate accessible design of visual, voice-based, and touch-based UIs.
    - Demonstrate understanding of laws and regulations applicable to accessible design.
    - Demonstrate understanding of what is appropriate and inappropriate high level of skill during interaction with individuals from diverse populations.
    - Analyze web pages and mobile apps for current standards of accessibility.
- Non-core
    - Find examples of how biases towards disability, race, and gender have historically, either intentionally or unintentionally, informed technology design and consider how those experiences (learnings?) might inform design.
    - Conceptualize user experience research to identify user needs and generate design insights.

Knowledge Unit 4: Evaluating the Design

- CS Core Topics
    - **Methods for evaluation with users:** formative (e.g. needs-finding and exploratory analysis) and summative assessment (e.g. functionality and usability testing); elements to evaluate (e.g., utility, efficiency, learnability, user satisfaction); understanding ethical approval requirements before engaging in user research.
- KA Core Topics
    - **Methods for evaluation with users:** qualitative (qualitative coding and thematic analysis), quantitative (statistical tests), and mixed methods (e.g., observation, think-aloud, interview, survey, experiment); presentation requirements (e.g., reports, personas); user-centered testing; heuristic evaluation; challenges and shortcomings to effective evaluation (e.g., sampling, generalization).
    - **Study planning:** how to set study goals, hypothesis design; approvals from Institutional Research Boards and ethics committees; how to pre-register a study; within-subjects vs. between-subjects design.
    - **Implications and impacts of design:** with respect to the environment, material, society, security, privacy, ethics, and broader impacts.
- Non-core Topics (including Emerging topics)
    - **Techniques and tools for quantitative analysis:** statistical packages; visualization tools; statistical tests (e.g., ANOVA, t-tests, post-hoc analysis, parametric vs non-parametric tests); data exploration and visual analytics; how to calculate effect size.
    - **Data management:** Data storage and data sharing (open science); sensitivity and identifiability.

Illustrative Learning Outcomes

- CS Core
    - Discuss the differences between formative and summative assessment and their role in evaluating design.
- KA Core
    - Select appropriate formative or summative evaluation methods at different points throughout the development of a design.
    - Discuss the benefits of using both qualitative and quantitative methods for evaluation.
    - Evaluate the implications and broader impacts of a given design.
    - Plan a usability evaluation for a given user interface, and justify its study goals, hypothesis design, and study design.
    - Conduct a usability evaluation of a given user interface and draw defensible conclusions given the study design.
- Non-core
    - Select and run appropriate statistical tests on provided study data to test for significance in the results.
    - Pre-register a study design, with planned statistical tests.

Knowledge Unit 5: System Design

- CS Core Topics
    - **Prototyping techniques and tools**: e.g., low-fidelity prototyping, rapid prototyping, throw-away prototyping, granularity of prototyping
    - **Design patterns:** iterative design, universal design, interaction design (e.g., data-driven design, event-driven design)
    - **Design constraints:** platforms, devices, resources
- KA Core Topics
    - **Design patterns and guidelines**: software architecture patterns, cross-platform design, synchronization considerations
    - **Design processes:** participatory design, co-design, double-diamond, convergence and divergence
    - **Interaction techniques:** input and output vectors (e.g., gesture, pose, touch, voice, force), graphical user interfaces, controllers, haptics, hardware design, error handling
    - **Visual UI design:** color, layout, Gestalt principles
- Non-core Topics (including Emerging topics)
    - **Immersive environments:** virtual reality, augmented reality, mixed reality, XR (which encompasses them), spatial audio.
    - **3D printing and fabrication**
    - **Asynchronous interaction models**
    - **Creativity support tools**
    - **Voice UI designs**

Illustrative Learning Outcomes

- CS Core
    - Propose system designs tailored to a specified appropriate mode of interaction.
    - Follow an iterative design and development process that incorporates understanding the user, developing an increment, evaluating the increment, and feeding those results into a subsequent iteration.
    - Explain the impact of changing constraints and design trade offs (e.g., hardware, user, security, etc.) on system design
- KA Core
    - Evaluate architectural design approaches in the context of project goals.
    - Identify synchronization challenges as part of the user experience in distributed environments.
    - Evaluate and compare the privacy implications behind different input techniques for a given scenario
    - Explain the rationale behind a UI design based on visual design principles
- Non-core

- Evaluate the privacy implications within a VR/AR/MR scenario

Knowledge Unit 6: HCI and SEP

- CS Core Topics
  - Universal & User-Centered Design
- KA Core Topics
  - Participatory & Inclusive Design Processes
  - Evaluating the Design: Implications and impacts of design: with respect to the environment, material, society, security, privacy, ethics, and broader impacts

Illustrative Learning Outcomes

- CS Core
  - Learning Outcome 1
  - Learning Outcome 2
- KA Core
  - Critique a recent example of a non-inclusive design choice, its societal implications, and propose potential design improvements
  - Evaluating the Design: Identify the implications and broader impacts of a given design.
- Non-core
  - Evaluate the privacy implications within a VR/AR/MR scenario

Professional Dispositions

- Adaptable
- Meticulous
- Empathetic:
- Team-oriented The successful HCI practitioner should focus on the success of the team.
- Creative: An HCI practitioner should design solutions that are informed by past practice, the needs of the audience, and HCI fundamentals. Creativity is required to blend these into something that solves the problem appropriately and elegantly.

Math Requirements

Required:

- Basic statistics to support the evaluation and interpretation of results, including central tendency, variability, frequency distribution

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Personal assistants (AI)
- Robotics (AI)
- Augmented intelligence (AI)
- Override in autonomous systems (AI)
- Data privacy (SEP)
- Inclusivity (SEP)
- Product Requirements (SE)
- Software Verification and Validation (SE)
- Interfacing with stakeholders (SE)
- Testing, design and communication tools (SE)

Competency Specifications

- **Task 1:** Compare and contrast different user-interface frameworks
- **Competency Statement:** Describe the strengths and weaknesses of particular frameworks for particular software design and development problems. This should demonstrate knowledge that design is goal-directed and involves tradeoffs in addition to knowledge of particular frameworks.
- **Competency area** Software
- **Competency unit** Requirements, Design, Documentation, Evaluation, Management
- **Required knowledge areas and knowledge units:** HCI / KU 1,4,5
    - HCI / KU 1,4,5
- **Required skill level:** Evaluate
- **Core level:**

- **Task 2:** Plan, document, and execute a usability study
- **Competency Statement:** Demonstrates knowledge of how feedback loops are used in a holistic design process and the ability to design a study that is appropriate to the system being studied, phase of the development process, and particular stakeholders under consideration.
- **Competency area** Application

- **Competency unit** Design, Documentation, Evaluation, Management, Consumer Acceptance, Adaptation to social issues, Improvement
- **Required knowledge areas and knowledge units:**
    - HCI / KU1,4
- **Required skill level:** Apply, Evaluate
- **Core level:**

- **Task 3:** Evaluate and provide recommendations to improve a user-facing system's usability, accessibility, and inclusivity
- **Competency Statement:** Demonstrate knowledge of various lenses for evaluating user-facing systems.
- **Competency area** Application
- **Competency unit** Evaluation, Documentation, Adaptation to social issues, Improvement
- **Required knowledge areas and knowledge units:**
    - HCI / KU1,3,4,5,6
- **Required skill level:** Evaluate
- **Core level:**

- **Task 4:** Collaborate on a cross-disciplinary team to support the design, evaluation and iteration of a user-facing system
- **Competency Statement:** Integrates the HCI knowledge units toward the goal of using software to solve real-world problems
- **Competency area** Software, Application
- **Competency unit** Requirements, Design, Development, Documentation, Evaluation, Management, Consumer Acceptance, Improvement
- **Required knowledge areas and knowledge units:**
    - HCI / KU1-6
- **Required skill level:** Develop
- **Core level:**

- **Task 5:** Given design requirements, prepare engineering specifications for a user interface
- **Competency Statement:** Develops software requirements specifications for user-facing interfaces in keeping with standards for usability, user needs, and user experience in accordance with a provided design.
- **Competency area:** Software, Application
- **Competency unit:** Requirements, Documentation
- **Required knowledge areas and knowledge units:**

    - HCI / K1,2,3
- **Required skill level:** Develop
- **Core level:**

- **Task 6:** Determine what aspects of an implementation require revision to support internationalization
- **Competency Statement:** Evaluates a system to identify culturally-relevant or language-relevant text, symbols, and patterns that may vary by locale.
- **Competency area:** Software / Systems / Application
- **Competency unit:** Design, Evaluation, Adaptation to social issues, Improvement
- **Required knowledge areas and knowledge units:**
    - HCI / K1,2,3,4
- **Required skill level:** Evaluate
- **Core level:**

- **Task 7:** Conduct a formative study to offer design strategy recommendations for a user-facing system
- **Competency Statement:** Designs, develops, and implements usability and user experience testing sessions to identify strengths, weaknesses, and opportunities for improvement within a user-facing system.
- **Competency area** Application, Theory
- **Competency unit:** Design, Testing, Consumer Acceptance, Improvement
- **Required knowledge areas and knowledge units:**
    - HCI / K 1,3,4
- **Required skill level:** Evaluate
- **Core level:**

- **Task 8:** Interpret and communicate feedback to support the iteration of a user-facing system under development
- **Competency Statement:** Demonstrate the ability to plan for collecting feedback, interpreting the feedback through an understanding of the language and priorities of the evaluator, and provide appropriate summaries and proposals for iterative development.

- **Competency area:** Software
- **Competency unit:** Requirements/Testing/Documentation/ Customer Acceptance/Adaptation to Social Issues/Improvement
- **Required knowledge areas and knowledge units:**
    - HCI /KU1-4
- **Required skill level:** Evaluate
- **Core level:**

- **Task 9:** Assess the ethical implications of deploying a user-facing system in a particular population
- **Competency Statement:** Demonstrate the ability to research a population and consider a proposed system's impact along multiple dimensions.
- **Competency area:** Application?
- **Competency unit** Requirements/Evaluation/Consumer Acceptance/Adaptation to social issues/Improvement
- **Required knowledge areas and knowledge units:**
    - HCI / KU1-4
- **Required skill level:** Evaluate
- **Core level:**

- **Task 10:** Prototype a user-interface based on a design specification and industry-standard guidelines
- **Competency Statement:** Develops a prototype for a user-facing interface of a software system according to a provided software design specification and in line with professional standards and expectations.
- **Competency area:** Software, Application, Theory?
- **Competency unit:** Requirements, / Design, Development
- **Required knowledge areas and knowledge units:**
    - HCI / K1,2,5
- **Required skill level:** Develop
- **Core level:**

Course Packaging Suggestions

Introduction to HCI for CS majors and minors, to include the following:

- HCI KU Understanding the User: 7 hours
- HCI KU Accountability and Responsibility in Design: 2 hours
- HCI KU Accessibility and Inclusive Design: 4 hours
- HCI KU Evaluating the Design: 3 hours
- HCI KU System Design: 10 hours
- HCI KU HCI and SEP: 2 hours

Pre-requisites:

- Agile software development

Skill statement: A student who completes this course should be able to describe user-centered design principles and apply them in the context of a small project.

Description: This sample course takes an integrative, project-oriented approach. The students learn HCI principles and apply them in a short, instructor-provided project in weeks 5-6. This motivates students to continue learning new concepts before embarking on a community-engaged final project in which they have to do the requirements analysis, design, implementation, and evaluation using rapid, iterative prototyping.

Weekly Topics:

1. Introduction to design (KU1,5)

2. Thinking, Acting, and Evaluating (KU1,4,5)

3. Memory and Mistakes (KU1,3,4)

4. Principles and Processes (KU5)

5-6. Integrating Design Processes and Software Development (KU5)

7. Design Thinking and Heuristic Evaluation (KU1,4)

8. Accessibility (KU2,3,6)9. Visual Design and Personas (KU1,3,4,5)

10. Final Project: Empathy and Identification (KU1-6)

11. Final Project: Ideation and Low-Fidelity Prototyping (KU1,4,5)

12-13: Final Project Implementation (KU5)

14: Final Project: Testing (KU4)

15: Final Project: Reporting (KU1-6)

Introduction to Data Visualization to include the following:

- GIT KU Visualization: 30 hours
- GIT KU Basic Rendering: 10 hours
- HCI, KU Understanding the User: 3 hours
- KUs DEI, Privacy, Ethics from SEP Knowledge Area: 4 hours

Pre-requisites:

- CS2
- Linear Algebra

Skill statement: A student who completes this course should understand how to select a dataset; ensure the data are accurate and appropriate; design, develop and test a visualization program that depicts the data and is usable.

Advanced Course: Usability Testing

- HCI KU 1 Understanding the User - hours
- HCI KU 2 Accountability and Responsibility in Design - hours
- HCI KU 3 Accessibilityand Inclusive Design - hours
- HCI KU 4 Evaluating the Design - hours
- HCI KU 5 System Design - hours
- HCI KU 6 HCI and SEP – hours

Pre-requisites:

- Introductory/Foundation courses in HCI
- Research methods, statistics

Description: This project-based course focuses on the formal evaluation of products. Topics include: usability test goal setting, recruitment of appropriate users, design of test tasks, design of the test environment, test plan development and implementation, analysis and interpretation of the results, and documentation and presentation of results and recommendations. Students will understand the techniques, procedures and protocols to apply in various situations for usability testing with users. Students will be able to design an appropriate evaluation plan, effectively conduct the usability test, collect data, and analyze results so that they can suggest improvements.

Suggested topics:

1. Planning the usability study
2. Defining goals
3. Study participants
4. Selecting tasks and creating scenarios
5. Deciding how to measure usability
6. Preparing test materials
7. Preparing the test environment
8. Conducting the pilot test
9. Conducting the test
10. Tabulating and analyzing data
11. Recommending changes
12. Communicating the results
13. Preparing the highlight tape
14. Changing the product and the process

Learning outcomes:

- Design an appropriate test plan
- Recruit appropriate participants
- Conduct a usability test
- Analyze results and recommend changes
- Present results
- Write a report documenting the recommended improvements

Committee

Chair: Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, USA

Members:

- Sherif Aly, The American University of Cairo, Egypt
- Jeremiah Blanchard, University of Florida, USA
- Zoya Bylinskii, Adobe Research, USA
- Paul Gestwicki, Ball State University, USA
- Susan Reiser, University of North Carolina at Asheville, USA
- Amanda M. Holland-Minkley, Washington and Jefferson College, USA
- Ajit Narayanan, Google, USA
- Nathalie Riche, Microsoft, USA
- Kristen Shinohara, Rochester Institute of Technology, USA
- Olivier St-Cyr, University of Toronto, Canada

Appendix: Core Topics and Skill Levels

Mathematical and Statistical Foundations (MSF)

Preamble

What is meant by mathematical foundations

A strong mathematical foundation remains a bedrock of computer science education, and mathematical and formal reasoning continue to play a significant role in computer science, in both theoretical and applied areas: developing algorithms, designing systems, modeling real-world phenomena, working with data. This Mathematical Foundations Knowledge Area – the successor to the ACM CS 2013 curriculum's "Discrete Structures" – seeks to identify the mathematical (inclusive of statistics) material that undergirds modern computer science. The bulk of the core mathematical topics of ACM CS 2013 remain; the change of name corresponds to a realization both that the broader name better describes some of the existing topics from 2013 and that some growing areas of computer science, such as artificial intelligence, machine learning, data science, and quantum computing, have continuous mathematics as their foundations too.

Methodology

The subcommittee's recommendations are informed by receiving input from the mathematical requirements in other Knowledge Areas (KAs), from the CS theory community, from various reports (example: Park City report on data science) and, critically, two surveys, one to faculty and one to industry practitioners. The first survey was issued to computer science faculty (with nearly 600 faculty responding) across a variety of institutional types and in various countries to obtain a snapshot of current practices in mathematical foundations and to solicit opinion on the importance of particular topics beyond the traditional discrete mathematics. The second survey was sent to industry employees (approximately 680 respondents) requesting their views on curricular topics and components.

Changes since CS 2013: While the traditional discrete math course remains a mainstay of computer science programs, what has changed is rising faculty concern about students' mathematical preparation and attitude coming into computer science. The most striking change from 2013, however, arises from considering the mathematical needs of the rapidly growing areas of artificial intelligence, machine learning, robotics, data science, and quantum computing. When survey's respondents, which included (self-identified) experts in these areas, rated the importance of a number of mathematical topics both for employment and graduate school, the top five broad content areas were: precalculus, calculus I, linear algebra, probability, and statistics.

Acknowledging some tensions

Faculty and students alike have strong opinions about how much and what math in CS. Generally, faculty, who themselves have strong theoretical training, are typically concerned about poor student preparation and motivation to learn math, while students complain about not seeing applications and wonder what any of the math has to do with the software jobs they seek. Even amongst faculty, there is recurring debate on whether calculus should be required of computer science students, accompanied by legitimate concern about the impact of calculus failure rates on computer science students. Yet, at the same time, the discipline has itself undergone a significant mathematical change: machine learning, robotics, data science, and quantum computing all demand a different kind of math than what's typically covered in a standard discrete structures course. The combination of changing mathematical demands and inadequate student preparation or motivation, in an environment of enrollment-driven strain on resources, has become a key challenge for CS departments.

Summary of recommendations

- ***Standardize the prerequisites to discrete math***. The faculty survey shows that institutional variation in discrete-math prerequisites distributes nearly evenly across algebra, precalculus and calculus, suggesting differing approaches to the mathematical maturity sought. Requiring precalculus appears to be a reasonable compromise, so that students come in with some degree of comfort with symbolic math and functions.
- ***Include applications in math courses***. Studies show that students are motivated when they see applications. We recommend including minor programming assignments or demonstrations of applications to increase student motivation. While computer science departments may not be able to insert such applications into courses offered by other departments, it is possible to include applications of math in the computer science courses that are co-scheduled with mathematical requirements, and to engage with textbook publishers to provide such material.
- ***Apply available resources to enable student success***. The subcommittee recommends that institutions adopt remedial options to ensure sufficient preparation without lowering standards in discrete mathematics. Theory courses can be moved further back in the curriculum to accommodate freshmen-year remediation, for example. And, where possible, institutions can avail of online tutoring systems (such as ALEKS) alongside regular coursework.
- ***Expand core mathematical requirements to meet the rising demand in new growth areas of computer science***. What is clear, looking forward to the next decade, is that exciting high-growth areas of computer science require a strong background in linear algebra, probability and statistics (preferably calculus-based). Accordingly, we recommend including as much of this material into the standard curriculum as possible,
- ***Send a clear message to students about mathematics while accommodating their individual circumstances***. Faculty and institutions are often under pressure to help every student succeed, many of whom struggle with math. While pathways, including computer science-adjacent degrees or tracks, can be created to steer students past math requirements towards software-focused careers, faculty should be equally direct in explaining the importance of sufficient mathematical preparation for graduate school and for the very topical areas that excite students.

- ***Adapt to institutional mission and student context***. Faculty often advise our students with a version of "the more math you take, the better" and, for plenty of students, particularly those bound for graduate school, that advice is sound; for others, it is unhelpful or even off-putting. Yet, institutions and students differ, with positive consequences for society. Some of these differences arise from the diversity of systems of higher education, within and across countries; some arise from varying goals for particular programs, for example adopting a more pre-professional curricular outlook or adopting a more foundational one. Liberal-arts colleges, for example, are severely constrained in how many technical courses they can require; yet, their student success validates their unique approach. Accordingly, we recommend that institutions creatively adapt these recommendations to their local context and strengths.

Core Hours

Knowledge Unit	CS Core	KA Core
Sets, Relations, and Functions	4	
Basic Logic	9	
Proof Techniques	10	1
Basics of Counting	5	
Graphs and Trees		1
Discrete Probability	6	2
Fundamentals of Calculus		TBD
Linear Algebra		TBD
Statistics		TBD
Total	**34**	**4**

Knowledge Units

MSF/PreCalculus
[Considered pre-requisites, not part of CS core]

Topics:

- Algebra: adding fractions, rules of exponents, solving linear or quadratic equations with one or two variables
- Functions: function notation, drawing and interpreting graphs of functions

- Exponentials and logarithms: a general familiarity with the functions and their graphs
- Geometry: distances between points, areas of common shapes
- Trigonometry: familiarity with basic trigonometric functions and the unit circle

MSF/Sets, Relations, and Functions
[4 CS Core hours]
Topics:

- Sets
  - Venn diagrams
  - Union, intersection, complement
  - Cartesian product
  - Power sets
  - Cardinality of finite sets
- Relations
  - Reflexivity, symmetry, transitivity
  - Equivalence relations, partial orders
- Functions
  - Surjections, injections, bijections
  - Inverses
  - Composition

Learning Outcomes:

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

MSF/Basic Logic
[9 CS Core hours]
Topics:

- Propositional logic (cross-reference: Propositional logic is also reviewed in AI/Knowledge Based Reasoning)
- Logical connectives
- Truth tables
- Normal forms (conjunctive and disjunctive)
- Validity of well-formed formula
- Propositional inference rules (concepts of modus ponens and modus tollens)
- Predicate logic
  - Universal and existential quantification
- Limitations of propositional and predicate logic (e.g., expressiveness issues)

Learning Outcomes:

1. Convert logical statements from informal language to propositional and predicate logic expressions.
2. Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae and computing normal forms.
3. Use the rules of inference to construct proofs in propositional and predicate logic.
4. 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.
5. 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.
6. Describe the strengths and limitations of propositional and predicate logic.

MSF/Proof Techniques
[10 CS Core hours, 1 KA Core hour]
Topics:

[CS Core]

- Notions of implication, equivalence, converse, inverse, contrapositive, negation, and contradiction
- The structure of mathematical proofs
- Direct proofs
- Disproving by counterexample
- Proof by contradiction
- Induction over natural numbers
- Structural induction
- Weak and strong induction (i.e., First and Second Principle of Induction)
- Recursive mathematical definitions

[KA Core]

- Well orderings

Learning Outcomes:

[CS Core]

1. Identify the proof technique used in a given proof.
2. Outline the basic structure of each proof technique (direct proof, proof by contradiction, and induction) described in this unit.
3. Apply each of the proof techniques (direct proof, proof by contradiction, and induction) correctly in the construction of a sound argument.
4. Determine which type of proof is best for a given problem.

5. Explain the parallels between ideas of mathematical and/or structural induction to recursion and recursively defined structures.
6. Explain the relationship between weak and strong induction and give examples of the appropriate use of each.

[KA Core]

7. State the well-ordering principle and its relationship to mathematical induction.

MSF/Basics of Counting
[5 CS Core hours]
Topics:

- Counting arguments
    - Set cardinality and counting
    - Sum and product rule
    - Inclusion-exclusion principle
    - Arithmetic and geometric progressions
- The pigeonhole principle
- Permutations and combinations
    - Basic definitions
    - Pascal's identity
    - The binomial theorem
- Solving recurrence relations (cross-reference: AL/Basic Analysis)
    - An example of a simple recurrence relation, such as Fibonacci numbers
    - Other examples, showing a variety of solutions
- Basic modular arithmetic

Learning Outcomes:

1. Apply counting arguments, including sum and product rules, inclusion-exclusion principle and arithmetic/geometric progressions.
2. Apply the pigeonhole principle in the context of a formal proof.
3. Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application.
4. 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).
5. Solve a variety of basic recurrence relations.
6. Analyze a problem to determine underlying recurrence relations.
7. Perform computations involving modular arithmetic.

MSF/Discrete Probability
[6 CS Core hours, 2 KA Core hour]
Topics:

[CS Core]

- Finite probability space, events
- Axioms of probability and probability measures
- Conditional probability, Bayes' theorem
- Independence
- Integer random variables (Bernoulli, binomial)
- Expectation, including Linearity of Expectation
-

[KA Core]

- Variance
- Conditional Independence

Learning Outcomes:

[CS Core]

1. Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance.
2. Differentiate between dependent and independent events.
3. Identify a case of the binomial distribution and compute a probability using that distribution.
4. Apply Bayes theorem to determine conditional probabilities in a problem.
5. Apply the tools of probability to solve problems such as the average case analysis of algorithms or analyzing hashing.

[KA Core]

6. Compute the variance for a given probability distribution.
7. Explain how events that are independent can be conditionally dependent (and vice-versa). Identify real-world examples of such cases.

MSF/Graphs and Trees
(cross-reference: AL/Fundamental Data Structures and Algorithms, especially with relation to graph traversal strategies)

Topics:

- Trees
    - Properties
    - Traversal strategies
- Undirected graphs

- Directed graphs
- Weighted graphs
- Spanning trees/forests
- Graph isomorphism

Learning Outcomes:

[CS Core]

1. Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of each type of graph/tree.
2. Demonstrate different traversal methods for trees and graphs, including pre, post, and in-order traversal of trees.
3. 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.
4. Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting.

[KA Core]

5. Explain how to construct a spanning tree of a graph.
6. Determine if two graphs are isomorphic.

MSF/Calculus

[KA Core]

Topics:

- Sequences, series, limits
- Single-variable derivatives: definition, computation rules (chain rule etc), derivatives of important functions, applications
- Single-variable integration: definition, computation rules, integrals of important functions, fundamental theorem of calculus, definite vs indefinite, applications (including in probability)
- Parametric formulation, polar representation
- Taylor series
- Multivariate calculus: partial derivatives, gradient, chain-rule, vector valued functions, applications to optimization, convexity, global vs local minima
- ODEs: definition, Euler method, applications to simulation

MSF/Linear Algebra

[KA Core]

Topics:

- Matrices, matrix-vector equation, geometric interpretation, geometric transformations with matrices
- Solving equations, row-reduction
- Linear independence, span, basis
- Orthogonality, projection, least-squares, orthogonal bases
- Linear combinations of polynomials, Bezier curves
- Eigenvectors and eigenvalues
- Applications to computer science: PCA, SVD, page-rank, graphics

MSF/Statistics

[KA Core]

Topics:

- Basic definitions and concepts: populations, samples, measures of central tendency, variance
- Univariate data: point estimation, confidence intervals
- Multivariate data: estimation, correlation, regression
- Data transformation: dimension reduction, smoothing
- Statistical models and algorithms

Professional Dispositions

- **Meticulous:** Students must pay attention to detail when applying math to problems
- **Persistent:** Students need to be persistent to both learn and apply math.

Shared Concepts and Crosscutting Themes

Shared concepts:

- MSF/Graphs and Trees is shared with AL/Fundamental Data Structures and Algorithms

Course Packaging Suggestions

Every department faces constraints in delivering content which precludes merely requiring a long list of courses covering every single desired topic. These constraints include content-area ownership, faculty size, student preparation, and limits on the number of departmental courses a curriculum can require.

We list below some options for mathematical foundations, combinations of which might best fit any particular institution:

- ***Traditional course offerings***. With this approach, a computer science department requires students to take math-department courses in the six broad mathematical areas listed above, for a total of 8 courses including Precalculus.
- ***A "Continuous Structures" analog of Discrete Structures***. Many computer science departments now offer courses that prepare students mathematically for AI and machine learning. Such courses can combine just enough calculus, optimization, linear algebra and probability; yet others may split linear algebra into its own course. These courses have the advantage of motivating students with computing applications, and including programming as pedagogy for mathematical concepts.
- ***Integration into application courses***. An application course such as machine learning can be spread across two courses, with the course sequence including the needed mathematical preparation taught just-in-time. This may have the advantage of mitigating turf issues and helping students see applications immediately after encountering math.
- ***Specific course adaptations***. For nearly a century, physics and engineering needs have driven the structure of calculus, linear algebra, and probability. Computer science departments can collaborate with their colleagues in math departments to restructure math-offered sections in these areas that are driven by computer science applications. For example, calculus could be reorganized along the lines described above, with all the computing needs fitted into two calculus courses, leaving later calculus for engineering and physics students.

Committee

Chair: Rahul Simha, George Washington University, Washington D.C., USA

Members:

- Richard L. Blumenthal, Regis University, Denver, CO, USA
- Marc Deisenroth, University College, London, UK
- Michael Goldweber, Xavier University, Cincinnati, OH, USA
- Cynthia Bailey Lee, Stanford University, Palo Alto, CA, USA
- David Liben-Nowell, Carleton College, Northfield, MN, USA
- Jodi Tims, Northeastern University, Boston, MA, USA

Networking and Communication (NC)

Preamble

Networking and communication play a central role in interconnected computer systems that are transforming the daily lives of billions of people. The public Internet provides connectivity for networked applications that serve ever-increasing numbers of individuals and organizations around the world. Complementing the public sector, major proprietary networks leverage their global footprints to support cost-effective distributed computing, storage, and content delivery. Advances in satellite networks expand connectivity to rural areas. Device-to-device communication underlies the emerging Internet of things.

This knowledge area deals with key concepts in networking and communication, as well as their representative instantiations in the Internet and other computer networks. Beside the basic principles of switching and layering, the area at its core provides knowledge on naming, addressing, reliability, error control, flow control, congestion control, domain hierarchy, routing, forwarding, modulation, encoding, framing, and access control. The area also covers knowledge units in network security and mobility, such as security threats, countermeasures, device-to-device communication, and multihop wireless networking. In addition to the fundamental principles, the area includes their specific realization in the Internet as well as hands-on skills in implementation of networking and communication concepts. Finally, the area comprises emerging topics such as network virtualization and quantum networking.

As the main learning outcome, learners develop a thorough understanding of the role and operation of networking and communication in networked computer systems. They learn how network structure and communication protocols affect behavior of distributed applications. The area educates on not only key principles but also their specific instantiations in the Internet and equips the student with hands-on implementation skills. While computer-system, networking, and communication technologies are advancing at a fast pace, the gained fundamental knowledge enables the student to readily apply the concepts in new technological settings.

Changes since CS 2013: Compared to the 2013 curricula, the knowledge area broadens its core tier-1 focus from the introduction and networked applications to include reliability support, routing, forwarding, and single-hop communication. Due to the enhanced core, learners acquire a deeper understanding of the impact that networking and communication have on behavior of distributed applications. Reflecting the increased importance of network security, the area adds a respective knowledge unit as a new elective. To track the advancing frontiers in networking and communication knowledge, the area replaces the elective unit on social networking with a new elective unit on emerging topics, such as middleboxes, virtualization, and quantum networking. Other changes consist of redistributing all topics from the old unit on resource allocation among other units, in order to resolve the unnecessary overlap between the knowledge units in the 2013 curricula.

Core Hours

Knowledge Units	CS Core	KA Core
Introduction	3	
Networked Applications	4	
Reliability Support		6
Routing And Forwarding		4
Single-Hop Communication		3
Mobility Support		4
Network Security		3
Emerging Topics		4
Total	**7**	**24**

Knowledge Units

NC/Introduction

Topics:

- Importance of networking in contemporary computing, and associated challenges.
- Organization of the Internet (e.g. users, Internet Service Providers, autonomous systems, content providers, content delivery networks).
- Switching techniques (e.g., circuit and packet).
- Layers and their roles (application, transport, network, datalink, and physical).
- Layering principles (e.g. encapsulation and hourglass model).
- Network elements (e.g. routers, switches, hubs, access points, and hosts).
- Basic queueing concepts (e.g. relationship with latency, congestion, service levels, etc.)

Learning Outcomes:

1. Articulate the organization of the Internet. [Familiarity]
2. List and define the appropriate network terminology. [Familiarity]
3. Describe the layered structure of a typical networked architecture. [Familiarity]
4. Identify the different types of complexity in a network (edges, core, etc.). [Familiarity]

NC/Networked Applications

Topics:

- Naming and address schemes (DNS, IP addresses, and Uniform Resource Identifiers).
- Distributed application paradigms (e.g. client/server, peer-to-peer, cloud, edge, and fog).
- Diversity of networked application demands (e.g. latency, bandwidth, and loss tolerance).
- An explanation of at least one application-layer protocol (e.g. HTTP, SMTP, and POP3).
- Interactions with TCP, UDP, and Socket APIs.

Learning Outcomes:

1. Define the principles of naming, addressing, resource location. [Familiarity]
2. Analyze the needs of specific networked application demands. [Familiarity].
3. Describe the details of one application layer protocol. [Familiarity].
4. Implement a simple client-server socket-based application. [Usage]

NC/Reliability Support

Topics:

- Unreliable delivery (e.g. UDP).
- Principles of reliability (e.g. delivery without loss, duplication, or out of order).
- Error control (e.g. retransmission, error correction).
- Flow control (e.g. stop and wait, window based).
- Congestion control (e.g. implicit and explicit congestion notification).
- TCP and performance issues (e.g. Tahoe, Reno, Vegas, Cubic, QUIC).

Learning Outcomes:

1. Describe the operation of reliable delivery protocols. [Familiarity]
2. List the factors that affect the performance of reliable delivery protocols. [Familiarity]
3. Describe some TCP reliability design issues [Familiarity].
4. Design and implement a simple reliable protocol. [Usage]

NC/Routing and Forwarding

Topics:

- Routing paradigms and hierarchy (e.g. intra/inter domain, centralized and decentralized, source routing, virtual circuits, QoS).
- Forwarding methods (e.g. forwarding tables and matching algorithms).
- IP and Scalability issues (e.g. NAT, CIDR, BGP, different versions of IP).

Learning Outcomes:

1. Describe various routing paradigms and hierarchies. [Familiarity]
2. Describe how packets are forwarded in an IP network. [Familiarity]
3. Describe how the Internet tackles scalability challenges. [Familiarity].

NC/Single-Hop Communication

Topics:

- Introduction to modulation, bandwidth, and communication media.
- Encoding and Framing.
- Medium Access Control (MAC) (e.g. random access and scheduled access).
- Ethernet.
- Switching.
- Local Area Network Topologies (e.g. data center networks).

Learning Outcomes:

1. Describe some basic aspects of modulation, bandwidth, and communication media. [Familiarity]
2. Describe in detail on a MAC protocol. [Familiarity]
3. Demonstrate understanding of encoding and framing solution tradeoffs. [Familiarity]
4. Describe details of the implementation of Ethernet [Familiarity]
5. Describe how switching works [Familiarity]
6. Describe one kind of a LAN topology [Familiarity]

NC/Network Security

Topics:

- General intro about security (Threats, vulnerabilities, and countermeasures).
- Network specific threats and attack types (e.g., denial of service, spoofing, sniffing and traffic redirection, man-in-the-middle, message integrity attacks, routing attacks, and traffic analysis)
- Countermeasures
    - Cryptography (e.g. SSL, symmetric/asymmetric).
    - Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, DMZ, Zero Trust Network Access, hyper network security, anonymous communication protocols, isolation)
    - Network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks, BGP Sec.

Learning Outcomes:

1. Describe some of the threat models of network security. [Familiarity]
2. Describe specific network-based countermeasures. [Familiarity]
3. Analyze various aspects of network security from a case study [Familiarity].

NC/Mobility

Topics:

- Principles of cellular communication (e.g. 4G, 5G).
- Principles of Wireless LANs (mainly 802.11).
- Device to device communication.
- Multihop wireless networks.
- Examples (e.g. ad hoc networks, opportunistic, delay tolerant).

Learning Outcomes:

1. Describe some aspects of cellular communication such as registration. [Familiarity]
2. Describe how 802.11 supports mobile users. [Familiarity]
3. Describe practical uses of device to device communication, as well as multihop. [Familiarity]
4. Describe one type of mobile network such as ad hoc. [Familiarity]

NC/Emerging topics

Topics:

- Middleboxes (e.g. filtering, deep packet inspection, load balancing, NAT, CDN).
- Virtualization (e.g. SDN, Data Center Networks).
- Quantum Networking (e.g. Intro to the domain, teleportation, security, Quantum Internet).

Learning Outcomes:

1. Describe the value of middleboxes in networks [Familiarity].
2. Describe the importance of Software Defined Networks [Familiarity]
3. Describe some of the added value achieved by using Quantum Networking [Familiarity]

Professional Dispositions

- Meticulous: In meeting being able to design networks and communication systems.
- Collaborative: Working in groups to achieve a common objective.
- Proactive: Anticipating changes in needs and acting upon them.
- Professional: Complying to the needs of the community in a responsible manner.
- Responsive: Acting swiftly to changes in needs.
- Adaptive: Making the required changes happen when needed.

Math Requirements

Required:

- Probability and Statistics
- Discrete Math (not sure this is needed)
- Simple queuing theory concepts.
- Fourier and trigonometric analysis for physical layer.

Shared Concepts and Crosscutting Themes

Course Packaging Suggestions

Coverage of the concepts of networking including but not limited to types of applications used by the network, reliability, routing and forwarding, single hop communication, security, and other emerging topics.

Note: both courses cover the same KU's but with different allocation of hours for each KU.

Introductory Course:

- NC/Introduction (9 hours)
- NC/Networked Applications (12 hours)
- NC/Reliability Support (6 hours)
- NC/Routing And Forwarding (4 hours)
- NC/Single-Hop Communication (3 hours)
- NC/Mobility Support (3 hours)
- NC/Network Security (3 hours)
- NC/Emerging Topics (2 hours)

Advanced Course:

- NC/Introduction (3 hours)
- NC/Networked Applications (4 hours)
- NC/Reliability Support (8 hours)
- NC/Routing And Forwarding (6 hours)
- NC/Single-Hop Communication (5 hours)
- NC/Mobility Support (5 hours)
- NC/Network Security (5 hours)
- NC/Emerging Topics (6 hours)

Competency Specifications

- **Task 1:** Write a white paper to explain stakeholder needs of a given networked environment.
- **Competency Statement:** Identify various stakeholders of a networked environment and explain their specific needs.
- **Competency area:** Systems
- **Competency unit:** Requirements / Documentation / Evaluation **Required knowledge areas and knowledge units:**
    - NC / Introduction
    - NC / Networked Applications
- **Required skill level:** Explain
- **Core level:**

- **Task 2:** Evaluate multiple network architectures and network elements to meet needs.
- **Competency Statement:** Identify various network architectures and associated network elements suitable for the problem at hand, and evaluate the most suitable approach to solve a given problem.
- **Competency area:** Systems / Application
- **Competency unit:** Requirements / Design / Documentation / Evaluation
- **Required knowledge areas and knowledge units:**
    - NC / Networked Applications
    - NC / Reliability Support
    - NC / Routing and Forwarding
    - NC / Security
    - NC / Mobility
- **Required skill level:** Explain / Evaluate
- **Core level:**

- **Task 3:** Develop a model to abstract a networked environment.
- **Competency Statement:** Use modeling techniques and tools to simplify a networked environment for subsequent development.
- **Competency area:** Systems / Application / Theory
- **Competency unit:** Requirements / Design / Documentation
- **Required knowledge areas and knowledge units:**
    - NC / Introduction
    - NC / Networked Applications
    - MS / <unknown yet>
- **Required skill level:** Develop
- **Core level:**

- **Task 4:** Develop a networking protocol.
- **Competency Statement:** Design and implement networking protocols that satisfy specific requirements, including various constraints during usage e.g. a simple file transfer protocol.
- **Competency area:** Systems / Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Documentation / Evaluation / Maintenance / Improvement
- **Required knowledge areas and knowledge units:**
    - NC / Networked Applications
    - NC / Reliability Support
    - NC / Routing and Forwarding
    - NC / Single Hop Communication
- **Required skill level:** Develop

- **Core level:**

- **Task 5:** Develop networked application.
- **Competency Statement:** Design and implement networked applications that satisfy specific requirements, including various constraints during usage.
- **Competency area:** Software / Application
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Documentation / Evaluation / Maintenance / Improvement
- **Required knowledge areas and knowledge units:**
    - NC / Networked Applications
    - NC / Security
    - NC / Mobility
- **Required skill level:** Develop
- **Core level:**

- **Task 6:** Evaluate the performance of a network, in specific latency, throughput, congestion, and various service levels.
- **Competency Statement:** Identify and evaluate various indicators of the performance of a network to suit specific needs.
- **Competency area:** Systems / Theory
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units:**
    - NC / Networked Applications
    - NC / Routing and Forwarding
- **Required skill level:** Evaluate
- **Core level:**

- **Task 7:** Defend the network from an ongoing distributed denial-of-service attack.
- **Competency Statement:** Identify the presence of an active security threat, the related vulnerabilities, and activate suitable countermeasures to defend the network from an ongoing attack of the given kind.
- **Competency area:** Systems / Application
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units (cross cutting):**
    - NC / Introduction
    - NC / Networked Applications
    - NC / Reliability Support
    - NC / Routing and Forwarding

    - NC / Security
    - NC / Emerging Topics
- **Required skill level:** Evaluate
- **Core level:**

- **Task 8:** Identify gray failures in a datacenter network.
- **Competency Statement:** Identify the presence of gray failures through multidimensional health monitoring.
- **Competency area:** Software / Systems / Application
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units (cross cutting):**
    - NC / Introduction
    - NC / Networked Applications
    - NC / Reliability Support
    - NC / Routing and Forwarding
    - NC / Security
    - NC / Emerging Topics
- **Required skill level:** Evaluate
- **Core level:**

- **Task 9:** Deploy and securely operate a network of wireless sensors.
- **Competency Statement:** Configure and deploy a given set of sensors in a networked environment to meet a certain need, and be able to operate them in a secure way.
- **Competency area:** Systems / Application
- **Competency unit:** Develop
- **Required knowledge areas and knowledge units (cross cutting):**
    - NC / Single Hop Communication
    - NC / Mobility
- **Required skill level:** Evaluate
- **Core level:**

- **Task 10:** Write a white paper to explain social, ethical, and professional issues governing the design and deployment of networked systems.
- **Competency Statement:** Identify various stakeholders and how social, ethical, and professional issues how the design and deployment of a given networking will affect them.
- **Competency area:** Systems / Theory
- **Competency unit:** Evaluation, Management, Adaptation to social issues.

- **Required knowledge areas and knowledge units:**
  - NC / Introduction
  - SEP / ???
- **Required skill level:** Evaluate
- **Core level:**

Committee

Chair: Sherif G. Aly - The American University in Cairo

Members:

- Khaled Harras: Carnegie Mellon University, Pittsburgh, USA
- Moustafa Youssef: The American University in Cairo, Cairo, Egypt
- Sergey Gorinsky: IMDEA Networks Institute
- Qiao Xiang: Xiamen University, China
- Alex (Xi) Chen: Huawei

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
NC	Introduction	• Importance of networking in contemporary computing, and associated challenges.	Explain	CS	3
		• Organization of the Internet ○ Users, ○ Internet Service Providers ○ Autonomous systems ○ Content providers ○ Content delivery networks	Explain		
		• Switching techniques ○ Circuit Switching ○ Packet Switching	Evaluate		
		• Layers and their roles. ○ Application ○ Transport ○ Network ○ Datalink ○ Physical	Explain		
		• Layering principles ○ Encapsulation	Explain		

		○ Hourglass model			
		● Network elements ○ Routers ○ Switches ○ Hubs ○ Access points ○ Hosts	Explain		
		● Basic queueing concepts ○ Relationship with latency ○ Relationship with Congestion ○ Relationship with Service levels	Explain		
NC	Networked Applications	● Naming and address schemes. ○ DNS ○ IP addresses ○ Uniform Resource Identifiers	Explain	CS	4
		● Distributed application paradigms ○ Client/server ○ Peer-to-peer ○ Cloud ○ Edge ○ Fog	Evaluate		
		● Diversity of networked application demands ○ Latency ○ Bandwidth ○ Loss tolerance	Explain		
		● Application-layer development using one or more protocols: ○ HTTP ○ SMTP ○ POP3	Develop		
		● Interactions with TCP, UDP, and Socket APIs.	Explain		
NC	Reliability Support	● Unreliable delivery ○ UDP ○ Other	Explain		
		● Principles of reliability ○ Delivery without loss ○ Duplication ○ Out of order	Develop	KA	6

		● Error control ○ Retransmission ○ Error correction	Evaluate		
		● Flow control ○ Stop and wait ○ Window based	Develop		
		● Congestion control ○ Implicit congestion notification ○ Explicit congestion notification	Explain		
		● TCP and performance issues ○ Tahoe ○ Reno ○ Vegas ○ Cubic ○ QUIC	Evaluate		
NC	Routing and Forwarding	● Routing paradigms and hierarchy ○ Intra/inter domain ○ Centralized and decentralized ○ Source routing ○ Virtual circuits ○ QoS	Evaluate	KA	4
		● Forwarding methods ○ Forwarding tables ○ Matching algorithms	Apply		
		● IP and Scalability issues ○ NAT ○ CIDR ○ BGP ○ Different versions of IP	Explain		
NC	Single Hop Communication	● Introduction to modulation, bandwidth, and communication media.	Explain	KA	3
		● Encoding and Framing.	Evaluate		
		● Medium Access Control (MAC) ○ Random access ○ Scheduled access	Evaluate		
		● Ethernet	Explain		
		● Switching	Apply		

		● Local Area Network Topologies (e.g. data center networks)	Explain		
NC	Network Security	● General intro about security [Shared with Security] ○ Threats ○ Vulnerabilities ○ Countermeasures	Explain	KA	4
		● Network specific threats and attack types [Shared with Security] ○ Denial of service ○ Spoofing ○ Sniffing ○ Traffic redirection ○ Man-in-the-middle ○ Message integrity attacks ○ Routing attacks ○ Traffic analysis	Explain		
		● Countermeasures [Shared with Security] ○ Cryptography (e.g. SSL, symmetric/asymmetric). ○ Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, DMZ, Zero Trust Network Access, hyper network security, anonymous communication protocols, isolation) ○ Network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks, BGP Sec.	Explain		
NC	Mobility	● Principles of cellular communication (e.g. 4G, 5G)	Explain	KA	3
		● Principles of Wireless LANs (mainly 802.11)	Explain		
		● Device to device communication [Shared with SPD]	Explain		
		● Multihop wireless networks	Explain		
		● Examples (e.g ad hoc networks, opportunistic, delay tolerant)	Explain		

NC	Emerging Topics	● Middleboxes (e.g. filtering, deep packet inspection, load balancing, NAT, CDN)	Explain	KA	4
		● Virtualization (e.g. SDN, Data Center Networks)	Explain		
		● Quantum Networking (e.g. Intro to the domain, teleportation, security, Quantum Internet)	Explain		

Operating Systems (OS)

Preamble

An operating system is the collection of services needed to safely interface the hardware with applications. Core topics focus on the mechanisms and policies needed to virtualize computation, memory, and I/O. Overarching themes that are reused at many levels in computer systems are well illustrated in operating systems (e.g. polling vs interrupts, caching, flexibility costs overhead, similar scheduling approaches to processes, page replacement, etc.).

A CS student needs to have a clear mental model of how a pipelined instruction executes to how data scope impacts memory location. Students can apply basic OS knowledge to domain-specific architectures (machine learning with GPUs or other parallelized systems, mobile devices, embedded systems, etc.). Since all software must leverage operating systems services, students can reason about the efficiency, required overhead and the tradeoffs inherent to any application or code implementation. The study of basic OS algorithms and approaches provides a context against which students can evaluate more advanced methods. Without an understanding of sandboxing, how programs are loaded into processes, and execution, students are at a disadvantage when understanding or evaluating vulnerabilities to vectors of attack.

The core of operating systems knowledge from CC2013 has been propagated from CC2013 to the updated knowledge area. Changes from CC2013 include moving of File systems knowledge (now called File Systems API and Implementation) and Device Management from elective to the core curriculum and Performance and Evaluation knowledge units to the Systems Fundamentals Knowledge area. The addition of persistent data storage and device I/O reflects the impact of file storage and device I/O limitations on the performance (e.g. parallel algorithms, etc.). To accommodate File Systems API and Implementation and Device Management as a CS Core knowledge unit, more advanced topics were moved from CS Core to KA Core. The Performance and Evaluation knowledge unit moved to Systems Fundamentals with the idea that performance and evaluation approaches for operating systems are mirrored at other levels and are best presented in this context.

Changes since CS 2013:

Core Hours

Knowledge Units	CS Core	KA Core
Role and Purpose of Operating Systems	2	
Principles of Operating System	2	
Concurrency	2	
Scheduling	1	
Process Model	1	
Memory Management	2	
Protection and Safety	2	
Device Management	1	
File Systems API and Implementation	2	
Virtualization		3
Real-time and Embedded Systems		2
Fault Tolerance		3
Total	**15**	**8**

Knowledge Units

OS/Role and purpose of the operating system

- CS Core Topics
    - Operating system as mediator between general purpose hardware and application-specific software

Example concepts: Operating system as an abstract virtual machine via an API)

- Universal operating system functions
  Example concepts:
  - Interfaces (process, user, device, etc)
  - Creation and execution of application specific software
  - Persistence of data
- Extended and/or specialized operating system functions (Example concepts: Embedded systems, Server types such as file, web, multimedia, boot loaders and boot security)
- Design issues (e.g. efficiency, robustness, flexibility, portability, security, compatibility, power, safety) Example concepts: Trade offs between error checking and performance, flexibility and performance, and security and performance
- Influences of security, networking, multimedia, parallel and distributed computing
- Overarching concern of security/protection: Neglecting to consider security at every layer creates an opportunity to inappropriately access resources.
  Example concepts:
  - Unauthorized access to files on an unencrypted drive can be achieved by moving the media to another computer,
  - Operating systems enforced security can be defeated by infiltrating the boot layer before the operating system is loaded and
  - Process isolation can be subverted by inadequate authorization checking at API boundaries

Illustrative Learning Outcomes

- CS Core
  - Understand the objectives and functions of modern operating systems
  - Evaluate the design issues in different usage scenarios (e.g. real time OS, mobile, server, etc)
  - Understand the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve
  - Understand how evolution and stability are desirable and mutually antagonistic in operating systems function

OS/Principles of operating systems

- CS Core Topics
  - Operating system software design and approaches such as Monolithic, Layered, Modular, Micro-kernel models and Unikernel
  - Abstractions, processes, and resources
  - Concept of system calls and links to application program interfaces (APIs)
    Example concepts:

        - Many system calls must be invoked to accomplish program application program requests
        - APIs (Win32, Java, Posix, etc) bridge the gap between highly redundant system calls and functions that are most aligned with the requests an application program would make
        - Approaches to syscall ABI (Linux "perma-stable" vs. breaking ABI every release).
    - The evolution of the link between hardware architecture and the operating system functions
    - Protection of resources means protecting some machine instructions/functions
      Example concepts
        - Applications cannot arbitrarily access memory locations or file storage device addresses
        - Protection of coprocessors and network devices
    - Leveraging interrupts from hardware level: service routines and implementations
      Example concepts
        - Timer interrupts for implementing timeslices
        - I/O interrupts for putting blocking threads to sleep without polling
    - Concept of user/system state and protection, transition to kernel mode using system calls
    - Mechanism for invoking of system calls, the corresponding mode and context switch and return from interrupt

Illustrative Learning Outcomes

- CS Core
    - Understand how the application of software design approaches to operating systems design/implementation (e.g. layered, modular, etc) affects the robustness and maintainability of an operating system
    - Categorize system calls by purpose
    - Understand dynamics of invoking a system call (passing parameters, mode change, etc)
    - Evaluate whether a function can be implemented in the application layer or can only be accomplished by system calls
    - Apply OS techniques for isolation, protection and throughput across OS functions (e.g. starvation similarities in process scheduling, disk request scheduling, semaphores, etc) and beyond
    - Understand how the separation into kernel and user mode affects safety and performance.
    - Understand the advantages and disadvantages of using interrupt processing in enabling multiprogramming
    - Analyze for potential threats to operating systems and the security features designed to guard against them

OS/Concurrency

- CS Core
    - Thread abstraction relative to concurrency
    - Race conditions, critical sections (role of interrupts if needed)
    - Deadlocks and starvation
    - Multiprocessor issues (spin-locks, reentrancy)
- KA Core Topics
    - Thread creation, states, structures
    - Thread APIs
    - Deadlocks and starvation (necessary conditions/mitigations)
    - Implementing thread safe code (semaphores, mutex locks, cond vars)
    - Race conditions in shared memory
- Non-core Topics (including Emerging topics)
    - Managing atomic access to OS objects Example concept: Big kernel lock vs. many small locks vs. lockless data structures like lists

Illustrative Learning Outcomes

- CS Core
    - Understand the advantages and disadvantages of concurrency as inseparable functions within the operating system framework
    - Understand how architecture level implementation results in concurrency problems including race conditions
    - Understand concurrency issues in multiprocessor systems
- KA Core
    - Understand the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each
    - Understand techniques for achieving synchronization in an operating system (e.g., describe how a semaphore can be implemented using OS primitives) including intra-concurrency control and use of hardware atomics
    - Accurately analyze code to identify race conditions and appropriate solutions for addressing race conditions

OS/Scheduling

- KA Core Topics
    - Preemptive and non-preemptive scheduling

    - Schedulers and policies Example concepts: First come, first serve, Shortest job first, Priority,Round Robin, and Multilevel
    - Concepts of SMP/multiprocessor scheduling and cache coherence
    - Timers (e.g. building many timers out of finite hardware timers)
- Non-core Topics (including Emerging topics)
    - Subtopics of operating systems such as energy-aware scheduling and real-time scheduling
    - Cooperative scheduling, such as Linux futexes and userland scheduling

Illustrative Learning Outcomes

- KA Core
    - Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes
    - Understand relationships between scheduling algorithms and application domains
    - Understand each types of processor schedulers such as short-term, medium-term, long-term, and I/O
    - Evaluate a problem or solution to determine appropriateness for asymmetric and/or symmetric multiprocessing.
    - Evaluate a problem or solution to determine appropriateness as a processes vs threads
    - Understand the need for preemption and deadline scheduling
- Non-core
    - Understand the ways that the logic embodied in scheduling algorithms is applicable to other operating systems mechanisms, such as first come first serve or priority to disk I/O, network scheduling, project scheduling, and problems beyond computing

OS/Process Model

- KA Core Topics
    - Processes and threads relative to virtualization-Protected memory, process state, memory isolation (see memory management) etc
    - Memory footprint/segmentation (stack, heap, etc)
    - Creating and loading executables and shared libraries
      Examples:
        - Dynamic linking, GOT, PLT
        - Structure of modern executable formats like ELF
    - Dispatching and context switching
    - Interprocess communication
        - Shared memory, message passing, signals, environment variables, etc

Illustrative Learning Outcomes

- KA Core

- Understand how processes and threads use concurrency features to virtualize control
- Understand reasons for using interrupts, dispatching, and context switching to support concurrency and virtualization in an operating system
- Understand the different states that a task may pass through and the data structures needed to support the management of many tasks
- Create executable using compilers and linkers from source code, shared libraries and object code
- Evaluate a software artifact and problem to determine appropriate use dynamic vs static shared libraries
- Understand the different ways of allocating memory to tasks, citing the relative merits of each
- Apply the appropriate interprocess communication mechanism for a specific purpose in a programmed software artifact

OS/Memory Management

- KA Core Topics
  - Review of physical memory, address translation and memory management hardware
  - Impact of memory hierarchy including cache concept, cache lookup, etc on operating system mechanisms and policy
    Example concepts:
    - CPU affinity and per-CPU caching is important for cache-friendliness and performance on modern processors
  - Logical and physical addressing, address space virtualization
  - Concepts of paging, page replacement, thrashing and allocation of pages and frames
  - Allocation/deallocation/storage techniques (algorithms and data structure) performance and flexibility
    Example concepts:
    - Arenas, slab allocators, free lists, size classes, heterogeneously sized pages (hugepages)
  - Memory Caching and cache coherence
  - Security mechanisms and concepts in memory mgmt including sandboxing, protection, isolation, and relevant vectors of attack
- Non-core Topics (including Emerging topics)
  - Virtual Memory: leveraging virtual memory hardware for OS services and efficiency

Illustrative Learning Outcomes

- KA Core
  - Explain memory hierarchy and cost-performance trade-offs
  - Summarize the principles of virtual memory as applied to caching and paging

    - Evaluate the trade-offs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed
    - Describe the reason for and use of cache memory (performance and proximity, how caches complicate isolation and VM abstraction)
    - Code/Develop efficient programs that consider the effects of page replacement and frame allocation on the performance of a process and the system in which it executes
- Non-core
    - Explain how hardware is utilized for efficient virtualization

OS/Protection and Safety (Overlap with Security TBD)

- CS Core Topics
    - Overview of operating system security mechanisms
    - Attacks and antagonism (scheduling, etc)
    - Review of major vulnerabilities in real operating systems
    - Operating systems mitigation strategies such as backups
- KA Core Topics
    - Policy/mechanism separation
    - Security methods and devices
      Example concepts:
        - Rings of protection (history from Multics to virtualized x86)
    - Protection, access control, and authentication

Illustrative Learning Outcomes

- CS Core
    - Understand the requirement for protection and security mechanisms in an operating systems
    - List and describe the attack vectors that leverage OS vulnerabilities
    - Understand the mechanisms available in an OS to control access to resources
- KA Core
    - Summarize the features and limitations of an operating system that impact protection and security

OS/Device management

- KA Core Topics
    - Buffering strategies
    - Direct Memory Access and Polled I/O, Memory-mapped I/O Example concept: DMA communication protocols (ring buffers etc)
    - Historical and contextual - Persistent storage device management (magnetic, SSD, etc)
- Non-core Topics (including Emerging topics)
    - Device interface abstractions, HALs

    - Device driver purpose, abstraction, implementation and testing challenges
    - High-level fault tolerance in device communication

Illustrative Learning Outcomes

- KA Core
    - Understand architecture level device control implementation and link relevant operating system mechanisms and policy (e.g. Buffering strategies, Direct memory access, etc)
    - Understand OS device management layers and the architecture (device controller, device driver, device abstraction, etc)
    - Understand the relationship between the physical hardware and the virtual devices maintained by the operating system
    - Explain I/O data buffering and describe strategies for implementing it
    - Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted
- Non-core
    - Describe the complexity and best practices for the creation of device drivers

OS/File Systems: API and Implementation (Historical significance but may play decreasing role moving forward)

- KA Core Topics
    - Concept of a file including Data, Metadata, Operations and Access-mode
    - File system mounting
    - File access control
    - File sharing
    - Basic file allocation methods including linked, allocation table, etc
    - File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (name, identified or metadata storage location)
    - Allocation/deallocation/storage techniques (algorithms and data structure) impact on performance and flexibility (i.e. Internal and external fragmentation and compaction)
    - Free space management such as using bit tables vs linking
    - Implementation of directories to segment and track file location

Illustrative Learning Outcomes

- KA Core
    - Understand the choices to be made in designing file systems
    - Evaluate different approaches to file organization, recognizing the strengths and weaknesses of each
    - Apply software constructs appropriately given knowledge of the file system implementation

OS/Advanced File systems

- KA Core Topics
    - File systems: partitioning, mount/unmount, virtual file systems
    - In-depth implementation techniques
    - Memory-mapped files
    - Special-purpose file systems
    - Naming, searching, access, backups
    - Journaling and log-structured file systems
- Non-core Topics (including Emerging topics)
    - Distributed file systems (e.g NAS, OSS, SAN, Cloud, etc)
    - Encrypted file systems
    - Fault tolerance (e.g. fsync and other things databases need to work correctly).

Illustrative Learning Outcomes

- KA Core
    - Understand how hardware developments have led to changes in the priorities for the design and the management of file systems
    - Map file abstractions to a list of relevant devices and interfaces
    - Identify and categorize different mount types
    - Understand specific file systems requirements and the specialize file systems features that meet those requirements
    - Understand the use of journaling and how log-structured file systems enhance fault tolerance
- Non-core
    - Understand purpose and complexity of distributed file systems
    - List examples of distributed file systems protocols
    - Understand mechanisms in file systems to improve fault tolerance

OS/Virtualization

- KA Core Topics
    - Using virtualization and isolation to achieve protection and predictable performance
    - Advanced paging and virtual memory
    - Virtual file systems and virtual devices
    - Containers
    - Thrashing
- Non-core Topics (including Emerging topics)
    - Types of virtualization (including Hardware/Software, OS, Server, Service, Network)
    - Portable virtualization; emulation vs. isolation
    - Cost of virtualization
    - VM and container escapes, dangers from a security perspective
    - Hypervisors

        - Hypervisor monitor w/o a host operating system
        - Host OS with kernel support for loading guests, e.g. QEMU KVM

Illustrative Learning Outcomes

- KA Core
    - Understand how hardware architecture provides support and efficiencies for virtualization
    - Understand difference between emulation and isolation
    - Evaluate virtualization trade-offs
- Non-core
    - Understand hypervisors and the need for them in conjunction with different types of hypervisors

OS/Real-time/embedded

- KA Core Topics
    - Process and task scheduling
    - Deadlines and real-time issues
    - Low-latency/soft real-time" vs "hard real time"
- Non-core Topics (including Emerging topics)
    - Memory/disk management requirements in a real-time environment
    - Failures, risks, and recovery
    - Special concerns in real-time systems (safety)

Illustrative Learning Outcomes

- KA Core
    - Understand what makes a system a real-time system
    - Understand latency and its sources in software systems and its characteristics.
    - Understand special concerns that real-time systems present, including risk, and how these concerns are addressed
- Non-core
    - Understand specific real time operating systems features and mechanisms

OS/Fault tolerance

- KA Core Topics
    - Reliable and available systems
    - Software and hardware approaches to address tolerance (RAID)
- Non-core Topics (including Emerging topics)
    - Spatial and temporal redundancy
    - Methods used to implement fault tolerance

    - Error identification and correction mechanisms
        - Checksumming of volatile memory in RAM
    - File system consistency check and recovery
    - Journaling and log-structured file systems
    - Use-cases for fault-tolerance (databases, safety-critical)
    - Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS's own services

Illustrative Learning Outcomes

- KA Core
    - Understand how operating system can facilitate fault tolerance, reliability, and availability
    - Understand the range of methods for implementing fault tolerance in an operating system
    - Understand how an operating system can continue functioning after a fault occurs
    - Understand the performance and flexibility trade offs that impact using fault tolerance
- Non-core
    - Describe operating systems fault tolerance issues and mechanisms in detail

OS/Social, ethical and professional

- KA Core Topics
    - Open source in operating systems
        - Identification of vulnerabilities in open source kernels
        - Open source guest operating systems
        - Open source host operating systems
        - Changes in monetization (paid vs free upgrades)
    - End-of-life issues with sunsetting operating systems

Illustrative Learning Outcomes

- KA Core
    - Understand advantages and disadvantages of finding and addressing bugs in open source kernels
    - Contextualize history and impact of Linux as an open source product
    - List complications with reliance on operating systems past end-of-life
    - Understand differences in finding and addressing bugs for various operating systems payment models

Professional Dispositions

- **Accountable** for decisions and their implications for security and performance
- **Meticulously** considers implication of OS mechanisms on any project

Math Requirements

Required:

- Discrete math

Shared Concepts and Crosscutting Themes

Shared concepts:

- Protection and Security (SEC)
- Distributed file systems (NC)
- Processes and threads relative to virtualization (AR)
- Memory hierarchy/Caching (AR)
- Buffering and Direct Memory Access, Polled and Memory-mapped I/O (AR)
- Device driver purpose, abstraction and implementation (AR)
- States and state diagrams (SF)
- Virtualization and isolation (SF)
- Reliable and available systems (SF)

Competency Specifications

- **Task 1:** Design and develop software that enables safe communication between processes
- **Competency Statement:** Apply communications mechanisms to safely communication between two processes
- **Competency area:** Software
- **Competency unit:** Design / Development / Testing
- **Required knowledge areas and knowledge units:**
    - OS / Role and Purpose of Operating Systems
    - OS / Principles of Operating Systems
    - OS / Concurrency
    - OS / Scheduling
    - OS / Process Model
    - OS / Memory Management

    - OS / Protect and Safety
    - PD / Communication
- **Required skill level:** Implement
- **Core level:**

- **Task 2:** Deploy an application component on an operating system runtime/virtualized operating system/container
- **Competency Statement:** Identify and mitigate potential problem with deployment; automate setup of deployment environment; set up monitoring of component execution
- **Competency area:** Systems
- **Competency unit:** Evaluation / Maintenance / Improvement
- **Required knowledge areas and knowledge units:**
    - OS / Role and Purpose of Operating Systems
    - OS / Principles of Operating Systems
    - OS / Concurrency
    - OS / Scheduling
    - OS / Process Model
    - OS / Memory Management
    - OS / Protect and Safety
    - AR / Assembly Level Machine Organization
- **Required skill level:** Apply
- **Core level:**

- **Task 3:** Deploy an application component on an operating system runtime/virtualized operating system/container
- **Competency Statement:** Identify and mitigate potential problem with deployment; automate setup of deployment environment; set up monitoring of component execution
- **Competency area** (select one or more)**:** Systems
- **Competency unit** (select one or more)**:** Evaluation / Maintenance / Improvement
- **Required knowledge areas and knowledge units:**
    - OS / Role and Purpose of Operating Systems
    - OS / Principles of Operating Systems
    - OS / Concurrency
    - OS / Process Model
    - OS / Memory Management

    - OS / Protect and Safety
- **Required skill level:** Apply
- **Core level:**

Course Packaging Suggestions

Introductory Course to include the following:

- Role and Purpose of Operating Systems- 3 hours
- Principles of Operating Systems- 3 hours
- Concurrency- 7 hours
- Scheduling- 3 hours
- Process Model- 3 hours
- Memory Management- 4 hours
- Protect and Safety- 4 hours
- Device Management- 2 hours
- File Systems API and Implementation- 2 hours
- Virtualization- 3 hours
- Advanced File Systems- 2 hours
- Real-time and Embedded Systems- 1 hours
- Fault Tolerance- 1 hours
- Social, Ethical and Professional topics- 4 hours

Pre-requisites:

- Assembly Level Machine Organization from Architecture
- Memory Management from Architecture
- Software Reliability from Architecture
- Interfacing and Communication from Architecture
- Functional Organization from Architecture

Skill statement: A student who completes this course should understand the impact and implications of operating system resource management in terms of performance and security. A student should understand and implement interprocess communication mechanisms safely. A student should differentiate between the use and evaluation of open source and/or proprietary operating systems. A student should understand virtualization as a feature of safe modern operating systems implementation.

Committee

Chair: Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA

Members:

- Renzo Davoli
- Avi Silberschatz
- Marcelo Pias, Federal University of Rio Grande (FURG), Brazil
- Mikey Goldweber, Xavier University, Cincinnati, USA
- Qiao Xiang, Xiamen University, China

Appendix: Core Topics and Skill Levels

	KU	Topics	Skill	Core	Hours
OS	Role and Purpose of Operating Systems	• Operating system as mediator between general purpose hardware and application-specific software [Overlap with PL] • Universal operating system functions • Extended and/or specialized operating system functions • Design issues (e.g. efficiency, robustness, flexibility, portability, security, compatibility, power, safety) • Influences of security, networking, multimedia, parallel and distributed computing • Overarching concern of security/protection: Neglecting to consider security at every layer creates an opportunity to inappropriately access resources.	Explain	CS	3
OS	Principles of Operating System	• Operating system software design and approaches • Abstractions, processes, files and resources	Explain	CS	3

		● Concept of system calls and links to application program interfaces (APIs) [Shared with AR] ● The evolution of the link between hardware architecture and the operating system functions [Shared with AR] ● Protection of resources means protecting some machine instructions/functions[Shared with AR] ● Leveraging interrupts from hardware level: service routines and implementations[Shared with AR] ● Concept of user/system state and protection, transition to kernel mode using system calls [Shared with AR] ● Mechanism for invoking of system calls, the corresponding mode and context switch and return from interrupt [Shared with AR]			
OS	Concurrency (non-core topics not listed)	● Thread abstraction relative to concurrency ● Race conditions, critical sections (role of interrupts if needed) ● Deadlocks and starvation ● Multiprocessor issues (spin-locks, reentrancy)	Explain	CS	3
		● Thread creation, states, structures ● Thread APIs ● Deadlocks and starvation (necessary conditions/mitigations) ● Implementing thread safe code (semaphores, mutex locks, cond vars) ● Race conditions in shared memory [Shared with PD]	Apply	KA	3
OS	Scheduling	● Preemptive and non-preemptive scheduling ● Timers (e.g. building many timers out of finite hardware timers).	Explain	KA	3

	(non-core topics not listed)	● Schedulers and policies ● Concepts of SMP/multiprocessor scheduling [Shared with AR]			
OS	Process Model	● Processes and threads relative to virtualization-Protected memory, process state, memory isolation, etc ● Memory footprint/segmentation (stack, heap, etc) ● Creating and loading executables and shared libraries ● Dispatching and context switching ● Interprocess communication	Explain	KA	3
OS	Memory Management (non-core topics not listed)	● Review of physical memory, address translation and memory management hardware ● Impact of memory hierarchy including cache concept, cache lookup, etc on operating system mechanisms and policy ● Logical and physical addressing ● Concepts of paging, page replacement, thrashing and allocation of pages and frames ● Allocation/deallocation/storage techniques (algorithms and data structure) performance and flexibility ● Memory Caching and cache coherence ● Security mechanisms and concepts in memory management including sandboxing, protection, isolation, and relevant vectors of attack	Explain	KA	4
OS	Protection and Safety (Overlap with Security)	● Overview of operating system security mechanisms ● Attacks and antagonism (scheduling, etc.) ● Review of major vulnerabilities in real operating systems ● Operating systems mitigation strategies such as backups	Apply	CS	3

		• Policy/mechanism separation • Security methods and devices • Protection, access control, and authentication	Apply	KA	1
OS	Device Management (non-core not listed) [Shared memory in AR]	• Buffering strategies • Direct Memory Access and Polled I/O, Memory-mapped I/O Historical and contextual - Persistent storage device management (magnetic, SSD, etc.)	Explain	KA	2
OS	File Systems API and Implementation (Historical significance but may play decreasing role moving forward)	• Concept of a file • File system mounting • File access control • File sharing • Basic file allocation methods • File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (name, identified or metadata storage location) • Allocation/deallocation/storage techniques (algorithms and data structure) impact on performance and flexibility • Free space management • Implementation of directories to segment and track file location	Explain	KA	2
OS	Advanced File Systems (non-core topics not listed)	• File systems: partitioning, mount/unmount, virtual file systems • In-depth implementation techniques • Memory-mapped files • Special-purpose file systems • Naming, searching, access, backups • Journaling and log-structured file systems	Explain	KA	2

OS	Virtualization(non-core topics not listed)	● Using virtualization and isolation to achieve protection and predictable performance ● Advanced paging and virtual memory ● Virtual file systems and virtual devices ● Thrashing ● Containers	Explain	KA	2
OS	Real-time and Embedded Systems (non-core topics not listed)	● Process and task scheduling ● Deadlines and real-time issues ● Low-latency/soft real-time" vs "hard real time" [shared with PL]	Explain	KA	1
OS	Fault Tolerance (non-core topics not listed)	● Reliable and available systems ● Software and hardware approaches to address tolerance (RAID)	Explain	KA	1
OS	Social, Ethical and Professional topics	● Open source in operating systems ● End-of-life issues with sunsetting operating systems [Covered in SE]	Explain	KA	4

Parallel and Distributed Computing (PDC)

Preamble

Parallel and distributed programming arranges and controls multiple computations occurring at the same time across different places. The ubiquity of parallelism and distribution are inevitable consequences of increasing numbers of gates in processors, processors in computers, and computers everywhere that may be used to improve performance compared to sequential programs, while also coping with the intrinsic interconnectedness of the world, and the possibility that some components or connections fail or misbehave. Parallel and distributed programming remove the restrictions of sequential programming that require computational steps to occur in a serial order in a single place, revealing further distinctions, techniques, and analyses applying at each layer of computing systems.

In most conventional usage, “parallel” programming focuses on arranging that multiple activities co-occur, “distributed” programming focuses on arranging that activities occur in different places, and “concurrent” programming focuses on interactions of ongoing activities with each other and the environment. However, all three terms may apply in most contexts. Parallelism generally implies some form of distribution because multiple activities occurring without sequential ordering constraints happen in multiple physical places (unless relying on context-switching schedulers or quantum effects). And conversely, actions in different places need not bear any particular sequential ordering with respect to each other in the absence of communication constraints. The focus of this KA is on contexts in which parallelism and distribution are explicitly introduced, but also apply when they are required by other constraints, such as when required data or services are intrinsically remote.

It can be challenging to teach and learn about an area that has evolved from being a diverse set of advanced topics into a central body of knowledge and practice, permeating almost every other aspect of computing. Nearly every problem with a sequential solution also admits parallel and/or distributed solutions; additional problems and solutions arise only in the context of existing concurrency. And nearly every application domain of parallel and distributed computing is a well-developed area of study and/or engineering too large to enumerate. Growth of the field has occurred irregularly across different subfields of computing, sometimes with different goals, terminology, practices, and associated courses, and sometimes masking the considerable overlap of basic ideas and skills that are the main focus of this KA.

Changes since CS 2013:

Core Hours

Knowledge Units	**CS Core hours**	**KA Core hours**
Programs and Execution	2	2
Communication	2	6
Coordination	2	6
Software Engineering	2	3
Algorithms and Application Domains	2	9
Total	**10**	**26**

CS Core topics span approaches to and aspects of parallel and distributed computing, but restrict coverage to those applying to nearly all of them. The main focus is on removing limitations of strictly sequential programming, revealing the essential structure and properties of parallel and distributed systems and software. Learning Outcomes include developing small programs (in a choice of several styles) with multiple activities and analyzing basic properties. The topics and hours do not include coverage of particular languages, tools, frameworks, systems, and platforms that would normally be included in any given course as a basis for implementing and evaluating concepts and skills. They also avoid reliance on specific choices that may vary widely (for example GPU programming vs cloud container deployment scripts), and include only brief mentions of related content more closely associated with Programming Languages, Computer Architecture, Networking, Security, and Systems KAs.

KA Core topics in each unit are of the form “One or more of the following” for a set of topics that extend associated core topics, These permit variation in coverage depending on the focus of any given course. As discussed further below, examples include a High Performance Computing course focusing on heterogeneous data parallelism mainly applied to linear algebra problems, a systems-oriented concurrency course focusing on shared-memory coordination mainly applied to resource management and middleware support, a distributed data processing course focusing on fault-tolerant data stores applied to web commerce applications, and so on. Depth of coverage of any KA Core subtopic is expected to vary according to course goals. For example, shared-memory coordination is a central topic in multicore programming, but much less so in most heterogeneous systems, and conversely for bulk data transfer. Similarly, fault tolerance is central to the design of distributed software, but much less so in most data-parallel applications.

Knowledge Units

PD/Programs and Execution [2 CS Core hours, 2 KA Core hours]

- Topics
    - [CS Core] Definitions and properties
        - Parallelizable actions
            - Including closures, functions, and services
            - Composite actions; sessions, tasks; scopes
            - Naming or identifying actions as parties (for example thread IDs)
            - Impact of granularity and overhead
        - Ordering among actions; happens-before relations
            - Relaxing ordering constraints permits nondeterministic execution of the series/parallel directed acyclic graphs representing program components
        - Independence: determining when ordering doesn't matter, in terms of commutativity, dependencies, preconditions
        - Consistency: Agreement about values and predicates; races, atomicity, consensus
        - Ensuring ordering when necessary in parallel programs
            - For example locking, safe publication
            - Communication among parties may impose ordering among their actions. As in: sending a message happens before receiving it
        - Places: Physical devices executing parallel actions (parties)
            - Hardware components, remote hosts
            - Places may support multiple parties (in sequence or schedule)
            - Parties may include external and human users
            - May include scheduling, time-slicing and emulation of multiple parallel actions by fewer processors
        - Faults arising from failures in parties or communication
            - Failures may be due to untrusted parties and protocols not under the control of the program or administrative domain
            - Degree of fault tolerance may be a design choice
    - [CS Core] Starting parallel actions
        - Placement: arranging that the action be performed (eventually) by a designated party
            - Details range from from hardwiring to configuration scripts, or relying on automated provisioning and management by platforms
            - Establishing communication and resource management
        - Procedural: Enabling multiple actions to start at a given program point

                - For example, starting new threads
                - May be contained in a scope ending when all complete
            - Reactive: Enabling upon an event
                - For example, installing event handlers
                - Less control of when actions begin or end
            - Dependent: Enabling upon completion of others
                - For example, sequencing
        - [KA Core] Underlying mappings and mechanisms. **One or more of:**
            - CPU data- and instruction-level- parallelism
            - SIMD and heterogeneous data parallelism
            - Multicore scheduled concurrency, tasks, actors
            - Clusters, clouds; elastic provisioning
            - Distributed systems with unbounded participants
            - Emerging technologies such as quantum computing and molecular computing
- Illustrative Learning Outcomes
    - [CS Core] Graphically show (as a dag) how to parallelize a compound numerical expression; for example a = (b+c) * (d + e).
    - [CS Core] Identify a race error in a given program
    - [CS Core] In a given context, explain the extent to which introducing parallelism in an otherwise sequential program would be expected to improve throughput and/or reduce latency, and how it may impact energy efficiency
    - [KA-Core] Write a function that efficiently counts events such as networking packet receptions
    - [KA-Core] Write a filter/map/reduce program in multiple styles
    - [KA-Core] Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client.

PD/Communication [2 CS core hours, 6 KA Core hours]

- Topics
    - [CS Core] Fundamentals
        - Media
            - Varieties: channels (message passing or IO), shared memory, heterogeneous, data stores
            - Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation
        - Channels
            - Explicit party-to-party communication; naming
            - APIs: sockets, architectural and language-based channel constructs
        - Memory

- Architectures in which parties directly communicate only with memory at given addresses
- Consistency: Bitwise atomicity limits, coherence, local ordering
- Memory hierarchies, locality: caches, latency, false-sharing
- Heterogeneous Memory using multiple memory stores, with explicit data transfer across them; for example, GPU local and shared memory, DMA
- Multiple layers of sharing domains, scopes and caches

■ Data Stores

- Cooperatively maintained structured data implementing maps, sets, and related ADTs
- Varieties: Owned, shared, sharded, replicated, immutable, versioned

○ [CS Core] Programming with communication

■ Using channel, socket, and/or remote procedure call APIs
■ Using shared memory constructs in a given language

○ [KA Core] Properties and Extensions. **One or more of**:

■ Media

- Topologies: Unicast, Multicast, Mailboxes, Switches; Routing
- Concurrency properties: Ordering, consistency, idempotency, overlapping with computation
- Reliability: transmission errors and drops.
- Data formats, marshaling
- Protocol design: progress guarantees, deadlocks
- Security: integrity, privacy, authentication, authorization.
- Performance Characteristics: Latency, Bandwidth (throughput), Contention (congestion), Responsiveness (liveness).
- Applications of Queuing Theory to model and predict performance

■ Channels

- Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control
- Program control for sending (usually procedural) vs receiving.(usually reactive or RPC-based)
- Formats, marshaling, validation, encryption, compressIon
- Multiplexing and demultiplexing in contexts with many relatively slow IO devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs.
- Formalization and analysis; for example using CSP

■ Memory

- Memory models: sequential and release/acquire consistency
- Memory management; including reclamation of shared data; reference counts and alternatives

            - Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays
            - Emulating shared memory: distributed shared memory, RDMA
        - Data Stores
            - Consistency: atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains,
            - Faults and partial failures; voting; protocols such as Paxos and Raft
            - Security and trust: Byzantine failures, proof of work and alternatives
- Illustrative Learning Outcomes
    - [CS Core] Determine whether shared memory or message passing would be preferable for a given application in a given context
    - [CS Core] Write a producer-consumer program in which one component generates numbers, and another computes their average. Measure speedups when the numbers are small scalars versus large multi-precision values.
    - [KA-Core] Write a program that distributes different segments of a data set to multiple workers, and collects results (for the simplest example, summing segments of an array).
    - [KA-Core] Write a parallel program that requests data from multiple sites, and summarizes them using some form of reduction
    - [KA-Core] Compare the performance of buffered versus unbuffered versions of a producer-consumer program
    - [KA-Core] Determine whether a given communication scheme provides sufficient security properties for a given usage
    - [KA-Core] Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent.
    - [KA-Core] Give an example of a scenario in which blocking message sends can deadlock.
    - [KA-Core] Describe at least one design technique for avoiding liveness failures in programs using multiple locks
    - [KA-Core] Write a program that illustrates memory-access or message reordering.
    - [KA-Core] Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates.
    - [KA-Core] Give an example of a scenario in which an attempted optimistic update may never complete.
    - [KA-Core] Modify a concurrent system to use a more scalable, reliable or available data store

PD/Coordination [2 CS core hours, 6 KA Core hours]

- Topics
    - [CS Core] Fundamentals
        - Dependent actions
            - Execution control when one activity's initiation or progress depends on actions of others
            - Completion-based: Barriers, joins

            - Data-enabled: Produce-Consumer designs
            - Condition-based: Polling, retrying, backoffs, helping, suspension, queueing, signaling, timeouts
            - Reactive: enabling and triggering continuations
        - Progress
            - Dependency cycles and deadlock; monotonicity of conditions
        - Atomicity
            - Atomic instructions, enforced local access orderings
            - Locks and mutual exclusion
            - Deadlock avoidance: ordering, coarsening, randomized retries; encapsulation via lock managers
            - Common errors: failing to lock or unlock when necessary, holding locks while invoking unknown operations, deadlock
    - [CS Core] Programming with coordination
        - Controlling termination
        - Using locks, barriers, and other synchronizers in a given language; maintaining liveness without introducing races
        - Using transactional APIs in a given framework
    - [KA Core] Properties and extensions. **One or more of:**
        - Progress
            - Properties including lock-free, wait-free, fairness, priority scheduling; interactions with consistency, reliability
            - Performance: contention, granularity, convoying, scaling
            - Non-blocking data structures and algorithms
        - Atomicity
            - Ownership and resource control
            - Lock variants: sequence locks, read-write locks; reentrancy; tickets
            - Transaction-based control: Optimistic and conservative
            - Distributed locking: reliability
        - Interaction with other forms of program control
            - Alternatives to barriers: Clocks; Counters, virtual clocks; Dataflow and continuations; Futures and RPC; Consensus-based, Gathering results with reducers and collectors
            - Speculation, selection, cancellation; observability and security consequences
            - Resource-based: Semaphores and condition variables
            - Control flow: Scheduling computations, Series-parallel loops with (possibly elected) leaders, Pipelines and Streams, nested parallelism.
            - Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting
- Illustrative Learning Outcomes
    - [CS Core] Show how to avoid or repair a race error in a given program

- [CS Core] Write a program that correctly terminates when all of a set of concurrent tasks have completed.
- [KA-Core] Write a function that efficiently counts events such as sensor inputs or networking packet receptions
- [KA-Core] Write a filter/map/reduce program in multiple styles
- [KA-Core] Write a program in which the termination of one set of parallel actions is followed by another
- [KA-Core] Write a program that speculatively searches for a solution by multiple activities, terminating others when one is found.
- [KA-Core] Write a program in which a numerical exception (such as divide by zero) in one activity causes termination of others
- [KA-Core] Write a program for multiple parties to agree upon the current time of day; discuss its limitations compared to protocols such as NTP
- [KA-Core] Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client

PD/Software Engineering [2 CS Core hours, 3 KA Core hours]

- Topics
  - [CS Core] Safety, liveness and performance requirements
    - Temporal logic constructs to express “always” and “eventually”
    - Metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements
  - [CS Core] Identifying, testing for, and repairing violations
    - Common forms of errors: failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), or termination (livelock)..
  - [KA Core] Specification and Evaluation. **One or more of:**
    - Formal Specification
      - Extensions of sequential requirements such as linearizability; protocol, session, and transactional specifications
      - Use of tools such as UML, TLA, program logics
      - Security: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting
    - Static Analysis
      - For correctness, throughput, latency, resources, energy
      - dag model analysis of algorithmic efficiency (work, span, critical paths)
    - Empirical Evaluation

- Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, continuous integration, continuous deployment, and test generators,
- Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of too many events, clients, threads.

■ Application domain specific analyses and evaluation techniques

- Illustrative Learning Outcomes
  - [CS Core] Revise a specification to enable parallelism and distribution without violating other essential properties or features
  - [CS Core] Explain why avoiding non-local side-effects is a common design goal
  - Specify a set of invariants that must hold at each bulk-parallel step of a computation
  - [CS Core] Write a test program that can reveal a data race error; for example, missing an update when two activities both try to increment a variable.
  - [KA-Core] Specify and measure behavior when a service is requested by too many clients
  - [KA-Core] Identify and repair a performance problem due to sequential bottlenecks
  - [KA-Core] Empirically compare throughput of two implementations of a common design (perhaps using an existing test harness framework).
  - [KA Core] Identify and repair a performance problem due to communication or data latency
  - [KA-Core] Identify and repair a performance problem due to communication or data latency
  - [KA-Core] Identify and repair a performance problem due to resource management overhead
  - [KA-Core] Identify and repair a reliability or availability problem

PD/Algorithms and Application Domains 2 CS Core hours, 9 KA Core hours]

- Topics
  - [CS Core] Expressing and implementing parallel and distributed algorithms
    - ■ Implementing concepts in given languages and frameworks to initiate activities (for example threads), use shared memory constructs, and channel, socket, and/or remote procedure call APIs
  - [CS Core] Survey:

Category	Typical Execution agents	Typical Communication mechanisms	Typical Algorithmic domains	Typical Engineering goals

Multicore	threads	Shared memory, Atomics, locks	Resource management, data processing	throughput, latency, energy
Reactive	Handlers, threads	IO Channels	Services, real-time	latency
Data parallel	GPU, SIMD, accelerators, hybrid	Heterogeneous memory	Linear algebra, graphics, data analysis	throughput, energy
Cluster	Managed hosts	Sockets, Message channels	Simulation, data analysis	throughput
Cloud	Provisioned hosts	Service APIs	Web applications	scalability
Open Distributed	Autonomous hosts	Sockets, Data stores	Fault tolerant data stores and services	reliability

    - [KA Core] Algorithmic Domains. **One of more of**:
        - Linear Algebra: Vector and Matrix operations, numerical precision/stability, applications in data analytics and machine learning
        - Data processing: sorting, searching and retrieval, concurrent data structures
        - Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics
        - Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms
        - Logic: SAT, concurrent logic programming
        - Graphics and computational geometry: Transforms, rendering, ray-tracing
        - Resource Management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts. Exclusive vs shared resources. Static, dynamic and elastic algorithms, Batching, prioritization, partitioning, decentralization via work-stealing and related techniques
        - Services: Implementing Web APIs, Electronic currency, transaction systems, multiplayer games.
- Illustrative Learning Outcomes
    - [CS Core] Implement a parallel/distributed component based on a known algorithm
    - [CS Core] Write a data-parallel program that for example computes the average of an array of numbers.

- [CS Core] Extend an event-driven sequential program by establishing a new activity in an event handler (for example a new thread in a GUI action handler).
- [CS Core] Improve the performance of a sequential component by introducing parallelism and/or distribution
- [CS Core] Choose among different parallel/distributed designs for components of a given system
- [KA-Core] Design, implement, analyze, and evaluate a component or application for X operating in a given context, where X is in one of the listed domains; for example a genetic algorithm for factory floor design.
- [KA-Core] Critique the design and implementation of an existing component or application, or one developed by classmates
- [KA-Core] Compare the performance and energy efficiency of multiple implementations of a similar design; for example multicore versus clustered versus GPU.

Professional Dispositions

-

Math Requirements

Shared Concepts and Crosscutting Themes

Shared Concepts:

Dependencies. Prerequisites for the CS core of the PD Knowledge Area include familiarity with:

- SDF (Software Development Fundamentals): programs vs executions, specifications vs implementations, variables, sequential control flow (conditionals, loops), procedure/function/method calls; arrays.
- MF (Math Fundamentals): logic, discrete structures including directed graphs, state machines. Plus at least minimal familiarity of linear algebra, differential equations
- PL (Programming languages) Event-driven, OO, functional styles and constructs
- SF (System Fundamentals): layered systems, Von Neumann architecture.

Prerequisites for KA core topics vary across options

Overlaps. Some of the following coverage could be placed in courses primarily focused on other KAs:

- SF (System Fundamentals): RPC, performance evaluation
- NC (Networking and communication): Protocols, APIs
- OS (Operating Systems) Concurrency, scheduling, fault tolerance
- PL (Programming languages) Interactions of parallelism with other forms of program control and semantics
- SE (Software Engineering) Requirements and analysis
- SEC (Security) TBD

There is also potential overlap with most other KAs, depending on coverage choices in Algorithms and Applications

Course Packaging Suggestions

The modest PD CS Core topic requirements do not constitute a specific course. They may be incorporated by extending coverage in one or more courses primarily devoted to one more of the following KAs:

- Programming Language (KU PL/PD)
- System Fundamentals (KU SF/xx)
- Operating Systems (KU OS/xx)
- Data Management (KU DM/xx)
- Networking (KU NC/xx)
- Computer Architecture (KU AR/xx)
- Algorithms (KU AL/xx)

Alternatively, or in addition, the PD CS core may serve as a basis for courses focussing on parallel and/or distributed computing. At one extreme, it is possible to offer a single broadly constructed course covering all PD KA Core topics to varying depths. At the other extreme, it is possible to infuse PD KA Core coverage across the curriculum by uniformly providing courses that cover parallel and distributed approaches alongside sequential ones for nearly every topic in computing. More conventional choices include courses that focus on one or a few categories (such as multicore or cluster), and algorithmic domains (such as linear algebra, or resource management). Such courses may go into further depth than listed in one or more KU, including development experience, but include only CS-Core-level coverage of other topics.

For the sake of a concrete example, here is a sample set of choices for a course mainly focusing on multicores:

1. Programs and Execution: CS Core plus KA Core on threads, tasks, instruction-level parallelism
2. Communication: CS Core plus KA core on multicore architectures, memory, concurrent data stores
3. Coordination: CS Core plus KA core on blocking and non-blocking synchronization, speculation, cancellation, futures, and divide-and-conquer data parallelism
4. Software Engineering: CS Core plus KA core on performance analysis
5. Algorithms and Applications: CS Core plus project-based KA Core coverage of data processing and resource management.

Competency Specifications

- **Task 1:** Implement a parallel/distributed component based on a known algorithm
- **Competency Statement:** Implement.parallel and distributed computing concepts that may be expressed in different ways across different languages and frameworks;
- **Competency area:** Software
- **Competency unit:** Development
- **Required knowledge areas and knowledge units:**
    - PDC / Algorithms and Applications
    - PL / Parallel and Distributed Computing
- **Required skill level:** Apply/Develop
- **Core level:** CS core

- **Task 2:** Improve the performance of a sequential application or component by introducing parallelism and/or distribution
- **Competency Statement:** Evaluate how and when parallelism and/or distribution can improve (or not improve) performance well enough to identify opportunities, as well as implement them and measure results
- **Competency area:** Software / Systems / Application
- **Competency unit:** Design/Development//Evaluation///Improvement
- **Required knowledge areas and knowledge units:**
    - PDC / Software Engineering
    - PL / Parallel and Distributed Computing
- **Required skill level:** Evaluate/Develop
- **Core level:** CS core

- **Task 3:** Revise a specification to enable parallelism and distribution without violating other essential properties or features
- **Competency Statement:** Ensure that relaxing sequential constraints and/or remotely communicating do not have unexpected consequences that break existing software.
  Competency area: Software / Systems / Application
- **Competency unit:** Design/Maintenance
- **Required knowledge areas and knowledge units:**
  - PDC / Software Engineering
  - PL / Parallel and Distributed Computing
- **Required skill level:** Explain//Evaluate
- **Core level:** CS core

- **Task 4:** Choose among different parallel/distributed designs for components of a given system
- **Competency Statement:** Evaluate the relative merits of, for example, d data parallel versus reactive designs are most applicable to problems at hand.
- **Competency area:** Software / Systems / Application
- **Competency unit:** Design//Evaluation
- **Required knowledge areas and knowledge units:**
  - PDC / Algorithms and Applications
  - PL / Parallel and Distributed Computing
- **Required skill level:**/Evaluate
- **Core level:** CS core

Committee

Chair: Doug Lea, State University of New York at Oswego, Oswego, USA

Members:

- Sherif Aly, American University of Cairo, Cairo, Egypt
- Michael Oudshoorn, High Point University, High Point, NC, USA
- Qiao Xiang, Xiamen University, China

- Dan Grossman, University of Washington, Seattle, USA
- Sebastian Burckhardt, Microsoft Research
- Vivek Sarkar, Georgia Tech, Atlanta, USA
- Maurice Herlihy, Brown University, Providence, USA
- Sheikh Ghafoor, Tennessee Tech, USA
- Chip Weems, University of Massachusetts, Amherst, USA

Contributors:

- Paul McKenney
- Peter Buhr

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
PD	Programs and Executions	Definitions and Properties ● Parallelizable actions ● Ordering among actions; happens-before relations ● Independence: determining when ordering doesn't matter in terms of commutativity, dependencies, preconditions ● Consistency: Agreement about values and predicates; races, atomicity, consensus ● Ensuring ordering when necessary in parallel programs ● Places: Physical devices executing parallel actions (parties) ● Faults arising from failures in parties or communication	Explain	CS	1
PD	Programs and Executions	Starting parallel actions ● Placement: arranging that the action be performed (eventually) by a designated party ● Procedural: Enabling multiple actions to start at a given program point ● Reactive: Enabling upon an event	Explain	CS	1

		● Dependent: Enabling upon completion of others			
PD	Programs and Executions	Underlying mappings and mechanisms. **One or more of:** ● CPU data- and instruction-level- parallelism ● SIMD and heterogeneous data parallelism ● Multicore scheduled concurrency, tasks, actors ● Clusters, clouds; elastic provisioning ● Distributed systems with unbounded participants ● Emerging technologies such as quantum computing and molecular computing	Explain	KA	2
PD	Communication	Fundamentals ● Media ○ Varieties: channels (message passing or IO), shared memory, heterogeneous, data stores ○ Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation ● Channels ○ Explicit party-to-party communication; naming ○ APIs: sockets, architectural and language-based channel constructs ● Memory ○ Architectures in which parties directly communicate only with memory at given addresses ○ Consistency: Bitwise atomicity limits, coherence, local ordering ○ Memory hierarchies, locality: caches, latency, false-sharing ○ Heterogeneous Memory using multiple memory stores, with explicit data transfer across them;	Explain	CS	2

		for example, GPU local and shared memory, DMA ○ Multiple layers of sharing domains, scopes and caches ● Data Stores ○ Cooperatively maintained structured data implementing maps, sets, and related ADTs ○ Varieties: Owned, shared, sharded, replicated, immutable, versioned			
PD	Communication	Programming with Communication ● Using channel, socket, and/or remote procedure APIs ● Using shared memory constructs in a given language	Develop	CS	1
PD	Communication	Properties and Extensions. **One or more of:** ● Media ○ Topologies: Unicast, Multicast, Mailboxes, Switches; Routing ○ Concurrency properties: Ordering, consistency, idempotency, overlapping with computation ○ Reliability: transmission errors and drops. ○ Data formats, marshaling ○ Protocol design: progress guarantees, deadlocks ○ Security: integrity, privacy, authentication, authorization. ○ Performance Characteristics: Latency, Bandwidth (throughput), Contention (congestion), Responsiveness (liveness). ○ Applications of Queuing Theory to model and predict performance ● Channels	Explain	KA	6

		○ Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control ○ Program control for sending (usually procedural) vs receiving.(usually reactive or RPC-based) ○ Formats, marshaling, validation, encryption, compressIon ○ Multiplexing and demultiplexing in contexts with many relatively slow IO devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs. ○ Formalization and analysis; for example using CSP ● Memory ○ Memory models: sequential and release/acquire consistency ○ Memory management; including reclamation of shared data; reference counts and alternatives ○ Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays ○ Emulating shared memory: distributed shared memory, RDMA ● Data Stores ○ Consistency: atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains, ○ Faults and partial failures; voting; protocols such as Paxos and Raft			

		○ Security and trust: Byzantine failures, proof of work and alternatives			
PD	Coordination	Fundamentals ● Dependent actions ○ Execution control when one activity's initiation or progress depends on actions of others ○ Completion-based: Barriers, joins ○ Data-enabled: Produce-Consumer designs ○ Condition-based: Polling, retrying, backoffs, helping, suspension, queueing, signaling, timeouts ○ Reactive: enabling and triggering continuations ● Progress ○ Dependency cycles and deadlock; monotonicity of conditions ● Atomicity ○ Atomic instructions, enforced local access orderings ○ Locks and mutual exclusion ○ Deadlock avoidance: ordering, coarsening, randomized retries; encapsulation via lock managers ○ Common errors: failing to lock or unlock when necessary, holding locks while invoking unknown operations, deadlock	Explain	CS	2
PD	Coordination	Programming with coordination ● Controlling termination ● Using locks, barriers, and other synchronizers in a given language; maintaining liveness without introducing races	Develop	CS	1

		● Using transactional APIs in a given framework			
PD	Coordination	Properties and extensions. **One or more of:** ● Progress ○ Properties including lock-free, wait-free, fairness, priority scheduling; interactions with consistency, reliability ○ Performance: contention, granularity, convoying, scaling ○ Non-blocking data structures and algorithms ● Atomicity ○ Ownership and resource control ○ Lock variants: sequence locks, read-write locks; reentrancy; tickets ○ Transaction-based control: Optimistic and conservative ○ Distributed locking: reliability ● Interaction with other forms of program control ○ Alternatives to barriers: Clocks; Counters, virtual clocks; Dataflow and continuations; Futures and RPC; Consensus-based, Gathering results with reducers and collectors ○ Speculation, selection, cancellation; observability and security consequences ○ Resource-based: Semaphores and condition variables ○ Control flow: Scheduling computations, Series-parallel loops with (possibly elected) leaders, Pipelines and Streams, nested parallelism.	Explain	KA	6

		○ Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting			
PD	Software Engineering	Safety, liveness and performance requirements ● Temporal logic constructs to express “always” and “eventually” ● Metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements	Explain	CS	1
PD	Software Engineering	Identifying, testing for, and repairing violations ● Common forms of errors: failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), or termination (livelock).	Evaluate	CS	1
PD	Software Engineering	Specification and Evaluation. **One or more of:** ● Formal Specification ○ Extensions of sequential requirements such as linearizability; protocol, session, and transactional specifications ○ Use of tools such as UML, TLA, program logics ○ Security: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting ● Static Analysis ○ For correctness, throughput, latency, resources, energy	Evaluate	KA	3

		○ dag model analysis of algorithmic efficiency (work, span, critical paths) ● Empirical Evaluation ○ Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, continuous integration, continuous deployment, and test generators, ○ Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of too many events, clients, threads. ● Application domain specific analyses and evaluation techniques			
PD	Algorithms and applications	Expressing and implementing parallel and distributed algorithms ● Implementing concepts in given languages and frameworks to initiate activities (for example threads), use shared memory constructs, and channel, socket, and/or remote procedure call APIs	Develop	CS	1
PD	Algorithms and applications	Survey of primary categories and algorithmic domains (listed below).	Explain	CS	1
PD	Algorithms and applications	Algorithmic Domains. **One or more of:** ● Linear Algebra: Vector and Matrix operations, numerical precision/stability, applications in data analytics and machine learning ● Data processing: sorting, searching and retrieval, concurrent data structures	Develop	KA	9

		● Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics ● Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms ● Logic: SAT, concurrent logic programming ● Graphics and computational geometry: Transforms, rendering, ray-tracing ● Resource Management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts. Exclusive vs shared resources. Static, dynamic and elastic algorithms, Batching, prioritization, partitioning, decentralization via work-stealing and related techniques ● Services: Implementing Web APIs, Electronic currency, transaction systems, multiplayer games.			

Security (SEC) – Work-In-Progress

Preamble

[As stated below, the Security Knowledge Area crosscuts and shares many concepts with the other CS2023 Knowledge Areas that are constantly being revised. Given these interdependencies, this version of the Security Knowledge Area should be considered as a work-in-progress version rather than a full beta release, which will need to be revised based on the other knowledge areas. Feedback is nevertheless sought on the approach being used to ensure the next version reflects the state of security within computer science in 2023.]

The world increasingly relies on computing infrastructure to support nearly every facet of modern critical infrastructure: transportation, communication, healthcare, education, energy generation and distribution, just to name a few. In recent years, with rampant attacks on and breaches of this critical infrastructure, it has become clearer that computer science graduates have an increased role in designing, implementing, and operating software systems that are secure and can keep information private.

In CS2023, the Security (SEC) Knowledge Area (KA) represents both crosscutting themes pervasive in all the other areas of CS2023, including Software Development Fundamentals, Data Management, Operating Systems, Networking and Communications, Parallel and Distributed Computing, Systems Fundamentals, and Artificial Intelligence. Consequently, a Security mindset needs to be incorporated into the overall ethos of computer science graduates so that security is inherent in all of their work product. (Also note the Security title was chosen as an umbrella term for this KA and includes concepts such as privacy, system design, and cryptography, which are included in the other KAs.)

The six crosscutting themes of cybersecurity, viewed with a computer science lens[2]: confidentiality, integrity, availability, risk, systems thinking, and adversarial thinking, are also relevant here. Of these, developing an adversarial thinking mindset is not typically covered in the other Computer Science Knowledge Areas (KAs), thus it is emphasized within the SEC core. Students also need to learn security concepts such as authentication, authorization, and non-repudiation. They also need to learn about system vulnerabilities and understand threats against software systems. As such, principles of protecting systems must be covered to complement system design principles covered in the SDF and SE KAs, including principles such as secure by design, privacy by design, or defense in depth. Another concept important in the SEC KA is the notion of assurance, which is an attestation that security mechanisms

[2] Joint Task Force on Cybersecurity Education. 2017. Cybersecurity Curricula 2017. Technical Report. ACM, IEEE-CS, AIS SIGSEC, and IFIP WG 11.8. https://doi.org/10.1145/3184594

comply with the security policies that have been defined for data, processes, and systems. With the increased use of computing systems and data sets in modern society, the issues of privacy, especially its technical aspects not covered in the Society, Ethics and Professionalism KA.

Changes since CS 2013: The Security KA is an "updated" name for CS2013's Information Assurance and Security (IAS) knowledge area. Since 2013, Information Assurance and Security has been rebranded as Cybersecurity, which has become a new computing discipline. Moreover, a Joint Task Force of the ACM, IEEE Computer Society, AIS and IFIP developed curricular guidelines, Cybersecurity 2017 (CSEC 2017) to reflect the new discipline. Therefore, the Security KA in CS2023 is informed by the notion of a disciplinary lens outlined in CSEC 2017 to focus on those aspects of security that are important for computer science students. CS2023's Security KA also incorporates recent developments in computer science for securing systems and improving privacy, building on CS2013's recognition of the pervasiveness of security in computer science. A Task Force has also been convened to develop the next decennial update of the CSEC 2017 guidelines: the focus in in CS2023 is on those aspects of security, privacy and related concepts relevant to the computer science discipline.

Core Hours

Several concepts in the Security KA are foundational elements of Computer Science curriculum for all students, as many graduates of undergraduate Computer Science programs will build software for the modern world where security is critical to the functioning of modern society. However, given the competing needs of other KAs and the limited amount of time available in most undergraduate programs, the philosophy here is to focus on security elements that students need to know that are not already required by other knowledge areas. As discussed below, the course packaging will show that most programs will be encouraged to go beyond the CS Core covered in this KA and the others listed below.

Therefore, the SEC KA CS Core Hours and KA Core Hours are shown in two parts.

1. CS Core Hours that the SEC area adds to the overall CS2023 curriculum that are not already included in the other KAs, and
2. CS Core Hours already included in the other KAs that are not meant to be duplicative, instead they are documentation that these are also important to the CS and KA Core Hours for the SEC KA. These numbers are shown in parentheses in the table below.

Knowledge Unit	CS Core	KA Core
Foundational Security	3	0

AL/Algorithms and Society	(2)	1
AL/Fundamental Data Structures and Algorithms*	(2)	2
AL/Cryptography*	1	4
AR/All Knowledge Units	1	3
DM/Data and Database Security	(2)	3
GIT/Animation and Immersion	0	2
HCI/Accountability and Responsibility in Design	(1)	2
MSF/Cryptography	(1)	(4)
NC/Network Security	(1)	2
OS/Role and purpose of OS	(1)	2
OS/Protection and Safety	(2)	2
PD/Communication	(2)	2
PD/Software Engineering	(1)	0
SEP/Privacy and Civil Liberties	(2)	2
SEP/Security Policies, Laws and Computer Crimes	(2)	2
SDF/Fundamental Programming Concepts	(2)	0
SE/Product Requirements	(1)	2
SE/Software Design	(1)	3

SE/Software Construction	(1)	2
SE/Software Verification and Validation	(1)	3
SPF/All Knowledge Units	(2)	3
SF/Systems Security	(2)	3
Total	4 (+ 29 included in other KAs)	45

Knowledge Units

SEC/Foundational Security

Topics:

- Crosscutting concepts within security: confidentiality, integrity, availability, risk, adversarial thinking, systems thinking
- Vulnerabilities, threats, and attack vectors
- Authentication and authorization, and access control techniques
- Concept of trust and trustworthiness
- Principles of protection, e.g., least privilege, open design, fail-safe defaults, defense in depth, layered defense
- Principles and practices of privacy
- Tensions between security, privacy, performance, and other design goals
- Legal issues
- Ethical considerations

Illustrative Learning Outcomes

- Design and develop a system that is secure against a set of identified threats
- Evaluate a system for trustworthiness
- Develop a system that incorporates various principles of security and evaluate it for its resilience to attacks
- Design and develop a system designed to protect individual privacy

SEC/Other Knowledge Units

Based on the right column of the table shown above, a set of Knowledge Units will be proposed here only if there is no natural home in the other Knowledge Areas. Such knowledge units, with their topics and illustrative learning outcomes, will be included in the next version of the Security KA.

KA and SEP

Security, with its associated topics such as privacy, are both providers and consumers of the SEP KA. While SEP focuses on broader social perspectives that inform the SEC KA, the SEC KA provides the "nuts and bolts" in computer science to ensure SEP can live up to its potential.

The SEP/Privacy and Civil Liberties Knowledge Unit covers the philosophical foundations of privacy rights, along with current and future legal directions. The Security KA covers the technical side of ensuring that privacy design and engineering principles can support privacy policies and regulations, as well as reduce or eliminate any adverse impact of privacy loss. Bias in algorithms and data (and its collection) need to be addressed within the Security KA.

The SEP/Security Policies, Laws and Computer Crimes Knowledge Unit covers the issues of computer crimes, malware, criminal hacking, cyber terrorism and more, all of which need to be understood with the Security KA to ensure appropriate technical solutions and safeguards can be implemented, especially for society at large that may not be familiar with computing.

Professional Dispositions

Possible dispositions typically included in computing by CC2020 are Perseverance, Inventive, Meticulous, Self-directed (e.g., self-learner), Collaborative, Proactive, Persistent, Professional, Responsible, Adaptable, Responsive, and Accountable. Although most professional dispositions are also desirable in the Security KA, the most desirable professional dispositions for this knowledge area are:

- *Meticulous*: Careful attention must be paid to details of the real world when developing a secure system to assure that every aspect of the system is protected. This requires meticulousness on the part of the student.

- *Self-directed* (e.g., self-learner): As the adversary is always going to look into newer ways of attacking systems and breaching data, students would need to constantly learn to keep up to date with the likely problems and attacks of tomorrow so that they are prepared to defend their systems and data.
- *Collaborative*: Most security and privacy breaches require collaboration with other personnel, especially in the face of a real-time attack. Students need to be able to work and rely on their teams to prevent breaches.
- *Responsible*: As society increasingly depends on computing infrastructure and information systems, students need to show responsibility when designing, developing, deploying, and maintaining secure systems.
- *Accountable*: The protection of systems and risk mitigation requires accountability if things go wrong. Therefore, future professionals need to know that they will be held accountable for security breaches and need to do their best in terms of design and implementation to prevent such breaches from occurring.

Math Requirements

Required:

- Discrete structures
- Group theory
- Linear algebra
- Number theory
- Probability
- Statistics

Shared Concepts and Crosscutting Themes

Shared concepts:

- Algorithms and Complexity
    - Cryptographic algorithms
- Architecture and Organization
    - Reverse engineering
- Artificial Intelligence
    - Machine learning models
- Data Management
    - Data security
- Graphics and Interactive Techniques
    - Privacy in XR systems
- Human-Computer Interaction

    - Usable security
- Mathematical Foundations
    - Cryptographic techniques
- Modeling
    - Access control models
- Networking and Communication
    - Secure networking protocols
- Operating Systems
    - Memory protection
- Parallel and Distributed Computing
    - Attacks due to race conditions
- Programming Languages
    - Secure compiler development
- Society, Ethics and Professionalism
    - Laws and ethics governing security and privacy
- Software Development Fundamentals
    - Defensive programming
- Software Engineering
    - Secure software engineering techniques
- Specialized Platform Development
    - Secure platform architectures
- Systems Fundamentals
    - Sandboxing techniques for isolation

Competency Specifications

- **Task 1**: Write a white paper to explain to co-workers what kinds of attacks an adversary might be able to attempt on the software being developed.
- **Competency area**: Software/Application
- **Competency unit**: Design/Development
- **Statement**: Apply adversarial thinking and systems thinking to each external interface being presented to the user.
- **Required knowledge areas and knowledge units**:
    - SEC/Foundational Security
    - DM/Data Security
    - SDF/Fundamental Programming Concepts
    - SE/Software Design
    - SE/Software Construction
    - SE/Software Verification and Validation
    - SEP/Privacy and Civil Liberties
    - SF/Systems Security

- **Required skill level**: Explain/Evaluate
- **Core level:**

Course Packaging Suggestions

The best way to make students aware of secure software development is to infuse security concepts into most of their coursework. However, it is likely that at least one course be offered that focuses on security so that students see the material in a holistic way. Although the breaches in computing are ongoing in the modern world, and all students should consider taking a course in security, it is unlikely that all computer science programs will be able to require such a course.

Some programs will find it easier to embed security concepts within existing courses, with Security topics blended in. For this to happen, the programs will need to understand how each of the other KAs, i.e., excluding the Security KA and the MSF KA. The guidance here to ensure every mention of security is emphasized when students are introduced to that particular concept, i.e., data security in a data management course, network security in a networking and communication courses, and so on.

Complementing this holistic could be a standalone course that offers the content listed in Foundational Security, along with some of the elective knowledge units. While the Security KA subcommittee would like to see coursework dedicated to security be in the required part of the curriculum, the members also realize that programs may not be able to do so. If so, offering such a course an elective is still important. As stated earlier, a standalone course on security may not be possible for many programs; if so, it becomes incumbent to integrate security concepts into other coursework.

Committee

Chair: Rajendra K. Raj, Rochester Institute of Technology, Rochester, NY, USA

Members:

- Vijay Anand, University of Missouri – St. Louis, MO, USA
- Diana Burley, American University, Washington, DC, USA
- Sherif Hazem, Central Bank of Egypt, Egypt
- Michele Maasberg, United States Naval Academy, Annapolis, MD, USA

- Sumita Mishra, Rochester Institute of Technology, Rochester, NY, USA
- Nicolas Sklavos, University of Patras, Patras, Greece
- Blair Taylor, Towson University, MD, USA
- Jim Whitmore, Distinguished IT Architect, USA

Society, Ethics and Professionalism (SEP)

Preamble

While technical issues are central to the computing curriculum, they do not constitute a complete educational program in the broader context. Students must also be exposed to the larger societal context of computing to develop an understanding of the relevant social, ethical, legal and professional issues. This need to incorporate the study of these non-technical issues into the ACM curriculum was formally recognized in 1991, as can be seen from the following excerpt from CS1991 [1]:

> *Undergraduates also need to understand the basic cultural, social, legal, and ethical issues inherent in the discipline of computing. They should understand where the discipline has been, where it is, and where it is heading. They should also understand their individual roles in this process, as well as appreciate the philosophical questions, technical problems, and aesthetic values that play an important part in the development of the discipline.*
>
> *Students also need to develop the ability to ask serious questions about the social impact of computing and to evaluate proposed answers to those questions. Future practitioners must be able to anticipate the impact of introducing a given product into a given environment. Will that product enhance or degrade the quality of life? What will the impact be upon individuals, groups, and institutions?*
>
> *Finally, students need to be aware of the basic legal rights of software and hardware vendors and users, and they also need to appreciate the ethical values that are the basis for those rights. Future practitioners must understand the responsibility that they will bear, and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools. All practitioners must make a long-term commitment to remaining current in their chosen specialties and in the discipline of computing as a whole.*

As technological advances continue to significantly impact the way we live and work, the critical importance of social and ethical issues and professional practice continues to increase in importance and consequence; new computer-based products and platforms pose ever more challenging problems each year. It is our students who will enter industry and academia with intentional regard for the identification and resolution of these issues.

Computer science educators may opt to deliver this material in stand-alone courses, integrated into traditional technical and theoretical courses, dedicated courses, some combination of both, or as special units as part of capstone, project, and professional practice courses. The material in this knowledge area is perhaps best covered through a combination of one required course along with aspects integrated in other technical courses. On the one hand, some topics in knowledge units listed as CS Core may not not readily lend themselves to being covered in other more traditional computer science courses. Without a standalone course, it is difficult to cover these topics appropriately. On the other hand, if social, ethical

and professional considerations are covered only in the standalone course and not in the context of other courses, it will reinforce the false notion that technical processes are void of these important issues. Because of this broad relevance, it is important that several traditional courses include aspects such as case studies that analyze the ethical, legal, social and professional considerations in the context of the technical subject matter of the course. Courses in areas such as software engineering, databases, computer graphics, computer networks, information assurance and security, and introduction to computing provide obvious context for analysis of such issues. However, an ethics-related module could be developed for almost any course in the curriculum. It would be explicitly against the spirit of these recommendations to have only a standalone course. Running through all of the issues in this area is the need to speak to the computing practitioner's responsibility to proactively address these issues by both ethical and technical actions. The ethical issues discussed in any course should be directly related to and arise naturally from the subject matter of that course. Examples include a discussion in a database course of the societal, ethical and professional aspects of data aggregation or data mining, or a discussion in the software engineering course of the potential conflicts between obligations to the customer and obligations to the user and others affected by their work. Programming assignments built around applications such as controlling the movement of a laser during eye surgery by non-computer scientists can help to address the social, ethical and professional impacts of computing. Computing faculty who are unfamiliar with the content and/or pedagogy of applied ethics are urged to take advantage of the considerable resources from ACM, IEEE-CS, SIGCAS (ACM Special Interest Group on Computers and Society), and other organizations.

It should be noted that the application of ethical analysis underlies every subsection of this Society, Ethics and Professionalism knowledge area in computing. The ACM Code of Ethics and Professional Conduct[1], the IEEE Code of Ethics[2], and the AAAI Code of Ethics and Professional Conduct[3] provide guidelines and case studies that serve as the basis for the conduct of our professional work. The General Moral Imperatives provide an understanding of our commitment to personal responsibility, professional conduct, and our leadership roles. It falls to computing educators to highlight the domain-specific role of these topics for our students, but programs should certainly be willing to lean heavily on complementary courses from the other humanities and social sciences.

Changes since CS 2013: Since 2013 all computing communities have become much more aware, and active, in areas of Justice, Equity, Diversity and Inclusion. All computing students deserve a just, equitable, diverse, and inclusive learning environment. However, computing students have a unique duty to ensure that when put to practice, their skills, knowledge, and competencies are applied in just, equitable, diverse, and inclusive ways. For these reasons, and as these issues are inherently a part of Society, Ethics, and Professionalism, a new knowledge unit has been added that addresses these issues.

Major changes from CS2013:

- Inclusion of SEP/Diversity, Equity, and Inclusion knowledge unit

[1] ACM/IEEE-CS Joint Curriculum Task Force, Computing Curricula 1991 (1991), ACM Press and IEEE Computer Society Press.

[1] www.acm.org/about/code-of-ethics

[2] https://www.ieee.org/about/corporate/governance/p7-8.html

[3] https://aaai.org/Conferences/code-of-ethics-and-conduct.php

Core Hours

Knowledge Units	CS Core	KA Core
Social Context	3	2
Methods for Ethical Analysis	2	1
Professional Ethics	2	2
Intellectual Property	1	1
Privacy and Civil Liberties	2	1
Professional Communication	2	1
Sustainability	1	1
History	0	1
Economies of Computing	0	1
Security Policies, Laws and Computer Crimes	2	1
Equity, Diversity and Inclusion	2	2
Total	**17**	**14**

Knowledge Units

SEP/Social Context

Computers and the Internet, perhaps more than any other technology, have transformed society over the past several decades, with dramatic increases in human productivity; an explosion of options for news, entertainment, and communication; and fundamental breakthroughs in almost every branch of science and engineering. Social Context provides the foundation for all other SEP knowledge units, especially Professional Ethics.

Topics:

[CS Core]

1. Social implications of computing in a hyper-networked world where the capabilities of artificial intelligence are rapidly evolving
2. Impact of social media and artificial intelligence on individual well-being, political ideology, and cultural ideology
3. Impact of involving computing technologies, particularly artificial intelligence, biometric technologies and algorithmic decision-making systems, in civic life (e.g. facial recognition technology, biometric tags, resource distribution algorithms, policing software)

[KA Core]

1. Growth and control of the internet, computing, and artificial intelligence
2. Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or underdeveloped countries
3. Accessibility issues, including legal requirements and dark patterns
4. Context-aware computing

Illustrative Learning Outcomes:

[CS Core]

1. Describe different ways that computer technology (networks, mobile computing, cloud computing) mediates social interaction at the personal and social group level.
2. Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability for diverse populations including under-represented populations and the disabled.
3. Interpret the social context of a given design and its implementation.
4. Evaluate the efficacy of a given design and implementation using empirical data.
5. Articulate the implications of social media use for different identities, cultures, and communities.

[KA Core]

1. Discuss the internet's role in facilitating communication between citizens, government, and each other.

2. Analyze the effects of reliance on computing in the implementation of democracy (e.g. delivery of social services, electronic voting).
3. Describe the impact of the under-representation of people from historically minoritized populations in the computing profession (e.g., industry culture, product diversity).
4. Explain the implications of context awareness in ubiquitous computing systems.
5. Access to the internet and computing technologies - how this affects different societies.
6. Discuss why/how internet access can be viewed as a human right.

SEP/Methods for Ethical Analysis

Ethical theories and principles are the foundations of ethical analysis because they are the viewpoints from which guidance can be obtained along the pathway to a decision. Each theory emphasizes different assumptions and methods for determining the ethicality of a given action. It is important for students to recognize that different decisions in different contexts may require different ethical theories to arrive at the best outcome, and what constitutes 'best' will be culturally determined within a given context and viewpoint. Applying methods for ethical analysis requires both an understanding of the underlying principles and assumptions guiding a given tool and an awareness of the social context for that decision. Traditional ethical frameworks as provided by western philosophy can be useful, but they are not all-inclusive. Effort must be taken to include decolonial, indigenous and historically marginalized ethical perspectives whenever possible. No theory will be universally applicable to all contexts, nor is any single ethical framework the 'best.' Engagement across various ethical schools of thought is important for students to develop the critical thinking needed in judiciously applying methods for ethical analysis of a given situation.

Topics:

[CS Core]

1. Avoiding fallacies and misrepresentation in argumentation
2. Ethical theories and decision-making (philosophical and social frameworks)
3. Recognition of the role culture plays in our understanding, adoption, design, and use of computing technology

[KA Core]

1. Professional checklists
2. Evaluation rubrics
3. Stakeholder analysis
4. Standpoint theory

Illustrative Learning Outcomes:

[CS Core]

1. Recognize and describe how a given cultural context impacts decision making.
2. Illustrate the use of example and analogy in ethical argument.
3. Analyze and avoid basic logical fallacies in an argument.
4. Analyze an argument to identify premises and conclusion.

[KA Core]

1. Evaluate all stakeholder positions in relation to their cultural context in a given situation.
2. Evaluate the potential for introducing or perpetuating ethical debt (deferred consideration of ethical impacts or implications) in technical decisions.

SEP/Professional Ethics

Computer ethics is a branch of practical philosophy that deals with how computing professionals should make decisions regarding professional and social conduct. There are three primary influences: 1) The individual's own personal ethical code, 2) Any informal or formal code of ethical behavior existing in the workplace, applicable licensures or certifications, and 3) Exposure to formal codes of ethics.

Topics:

[CS Core]

1. Community values and the laws by which we live
2. The nature of professionalism including care, attention and discipline, fiduciary responsibility, and mentoring
3. Keeping up-to-date as a computing professional in terms of familiarity, tools, skills, legal and professional frameworks as well as the ability to self-assess and progress in the computing field
4. Professional certification, codes of ethics, conduct, and practice, such as the ACM/IEEE-CS, SE, AITP, IFIP and international societies
5. Accountability, responsibility and liability (e.g. software correctness, reliability and safety, as well as ethical confidentiality of cybersecurity professionals)
6. Introduction to theories describing the human creation and use of technology including instrumentalism, sociology of technological systems, disability justice, neutrality thesis, pragmatism, utilitarianism, and decolonial theories
7. Develop strategies for recognizing and reporting designs, systems, software, and professional conduct (or their outcomes) that may violate law or professional codes of ethics.

[KA Core]

1. The role of the computing professional in public policy
2. Maintaining awareness of consequences
3. Ethical dissent and whistle-blowing
4. The relationship between regional culture and ethical dilemmas
5. Dealing with harassment and discrimination
6. Forms of professional credentialing
7. Acceptable use policies for computing in the workplace
8. Ergonomics and healthy computing environments
9. Time to market and cost considerations versus quality professional standards

Illustrative Learning Outcomes:

[CS Core]

1. Identify ethical issues that arise in software design, development practices, and software deployment

2. Determine how to address ethical issues.
3. Explain the ethical responsibility of ensuring software correctness, reliability and safety including from where this responsibility arises (e.g. ACM/IEEE/AAAI Codes of Ethics, laws and regulations, organizational policies).
4. Describe the mechanisms that typically exist for a professional to keep up-to-date in ethical matters.
5. Describe the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making.
6. Analyze a global computing issue, observing the role of professionals and government officials in managing this problem.
7. Describe the philosophical underpinnings of the creation of software and how it informs our decisions of how to use it.

[KA Core]

1. Describe ways in which professionals may contribute to public policy.
2. Describe the consequences of inappropriate professional behavior.
3. Be familiar with whistleblowing and have access to knowledge to guide one through an incident.
4. Provide examples of how regional culture interplays with ethical dilemmas.
5. Discuss forms of harassment and discrimination and avenues of assistance.
6. Examine various forms of professional credentialing.
7. Explain the relationship between ergonomics in computing environments and people's health.
8. Describe issues associated with industries' push to focus on time to market versus enforcing quality professional standards.

SEP/Intellectual Property

Intellectual property refers to a range of intangible rights of ownership in any product of the human intellect, such as a software program. Laws which vary by country, provide different methods for protecting these rights of ownership based on their type. There are essentially four types of intellectual property rights relevant to software: patents, copyrights, trade secrets and trademarks. Each affords a different type of legal protection.

Topics:

[CS Core]

1. Philosophical foundations of intellectual property
2. Intellectual property rights
3. Intangible digital intellectual property (IDIP)
4. Legal foundations for intellectual property protection

[KA Core]

1. Digital rights management
2. Copyrights, patents, trade secrets, trademarks

3. Plagiarism
4. Foundations of the open source movement
5. Software piracy

Illustrative Learning Outcomes:

[CS Core]

1. Discuss the philosophical bases of intellectual property in an appropriate context (e.g. country, etc.).
2. Describe legislation aimed at digital copyright infringements.
3. Critique legislation aimed at digital copyright infringements.
4. Identify contemporary examples of intangible digital intellectual property.
5. Justify uses of copyrighted materials.
6. Evaluate the ethical issues inherent in various plagiarism detection mechanisms.

[KA Core]

1. Interpret the intent and implementation of software licensing.
2. Weigh the conflicting issues involved in securing software patents.
3. Characterize and contrast the concepts of copyright, patenting and trademarks.
4. Explain the rationale for the legal protection of intellectual property in the appropriate context (e.g. country, etc.).
5. Identify the goals of the open source movement.
6. Characterize the global nature of software piracy.

SEP/Privacy and Civil Liberties

Electronic information sharing highlights the need to balance privacy protections with information access. The ease of digital access to many types of data makes privacy rights and civil liberties more complex, differing among the variety of cultures worldwide.

Topics:

[CS Core]

1. Privacy implications of widespread data collection for transactional databases, data warehouses, surveillance systems, and cloud computing
2. Ramifications of differential privacy
3. Technology-based solutions for privacy protection
4. Civil liberties and cultural differences

[KA Core]

1. Philosophical foundations of privacy rights
2. Legal foundations of privacy protection in relevant jurisdictions
3. Privacy legislation in areas of practice

4. Freedom of expression and its limitations

Illustrative Learning Outcomes:

[CS Core]

1. Evaluate solutions to privacy threats in transactional databases and data warehouses.
2. Describe the role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile computing).
3. Describe the ramifications of differential privacy.
4. Investigate the impact of technological solutions to privacy and security issues (e.g. Differential Privacy).

[KA Core]

1. Discuss the philosophical basis for the legal protection of personal privacy in an appropriate context (e.g. country, etc.).
2. Critique the intent, potential value and implementation of various forms of privacy legislation.
3. Identify strategies to enable appropriate freedom of expression.

SEP/Professional Communication

Professional communication conveys information to various audiences who may have very different goals and needs for that information. Effective professional communication of technical information is rarely an inherited gift, but rather needs to be taught in context throughout the undergraduate curriculum. Like most skills, it requires practice.

Topics:

[CS Core]

1. Interpreting, summarizing, and synthesizing technical material, including source code and documentation
2. Writing effective technical documentation and materials (tutorials, reference materials, API documentation)
3. Identifying, describing, and employing (clear, polite, concise) oral, written, and electronic team and group communication.
4. Understanding and enacting awareness of audience in communication by communicating effectively with different customers, stakeholders, and leadership
5. Utilizing collaboration tools
6. Recognizing and avoiding the use of rhetorical fallacies when resolving technical disputes
7. Understanding accessibility and inclusivity requirements for addressing professional audiences

[KA Core]

1. Demonstrate cultural competence in written and verbal communication
2. Using synthesis to concisely and accurately convey tradeoffs in competing values driving software projects including technology, structure/process, quality, people, market and financial

3. Use writing to solve problems or make recommendations in the workplace, such as raising ethical concerns or addressing accessibility issues

Illustrative Learning Outcomes:

[CS Core]

1. Understand the importance of writing concise and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references.
2. Evaluate written technical documentation for technical accuracy, concision, lack of ambiguity, and awareness of audience.
3. Develop and deliver an audience aware, accessible, and organized formal presentation.
4. Plan interactions (e.g. virtual, face-to-face, shared documents) with others in ways that invite inclusive participation, model respectful consideration of others' contributions, and explicitly value diversity of ideas.
5. Recognize and describe qualities of effective communication (e.g. virtual, face-to-face, shared documents).

[KA Core]

1. Discuss ways to influence performance and results in diverse and cross-cultural teams.
2. Evaluate personal strengths and weaknesses to work remotely as part of a team drawing from diverse backgrounds and experiences.

SEP/Sustainability

Sustainability is defined by the United Nations as "development that meets the needs of the present without compromising the ability of future generations to meet their own needs." Alternatively it is the "balance between the environment, equity and economy." (UCLA Sustainability). As computing extends into more and more aspects of human existence, we are already seeing estimates that 10% of global electricity usage is spent on computing, and that percentage seems prone to continue growing. Further, electronics contribute individually to demand for rare earth elements, mineral extraction, and countless e-waste concerns. Students should be prepared to engage with computing with a background that recognizes these global and environmental costs, and their potential long term effects on the environment and local communities.

Topics:

[CS Core]

1. Being a sustainable practitioner by taking into consideration environmental, social, and cultural impacts of implementation decisions (e.g. algorithmic bias/outcomes, , economic viability, and resource consumption)
2. Explore local/regional/global social and environmental impacts of computing systems use and disposal (e-waste)
3. Discuss the tradeoffs involved in proof-of-work and proof-of-stake algorithms

[KA Core]

1. Guidelines for sustainable design standards
2. Systemic effects of complex computer-mediated phenomena (e.g. social media, offshoring, remote work)
3. Pervasive computing: Information processing that has been integrated into everyday objects and activities, such as smart energy systems, social networking and feedback systems to promote sustainable behavior, transportation, environmental monitoring, citizen science and activism
4. Conduct research on applications of computing to environmental issues, such as energy, pollution, resource usage, recycling and reuse, food management / production, and others
5. How the sustainability of software systems are interdependent with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g. market forces, government policies)

Illustrative Learning Outcomes:

[CS Core]

1. Identify ways to be a sustainable practitioner.
2. For any given project (software artifact, hardware, etc) enumerate the environmental impacts of its deployment. (e.g. energy consumption, contribution to e-waste, impact of manufacturing)
3. Illustrate global social and environmental impacts of computer use and disposal (e-waste).
4. List the sustainable effects of modern practices and activities such as telecommuting, web shopping, or cryptocurrency mining.

[KA Core]

1. Describe the environmental impacts of design choices within the field of computing that relate to algorithm design, operating system design, networking design, database design, etc.
2. Investigate the social and environmental impacts of new system designs.
3. Identify guidelines for sustainable IT design or deployment.
4. Investigate pervasive computing in areas such as smart energy systems, social networking, transportation, agriculture, supply-chain systems, environmental monitoring and citizen activism.
5. Assess computing applications in respect to environmental issues (e.g. energy, pollution, resource usage, recycling and reuse, food management and production).

SEP/History

This history of computing is taught to provide a sense of how the rapid change in computing impacts society on a global scale. It is often taught in context with foundational concepts, such as system fundamentals and software development fundamentals. History is important because it provides a mechanism for understanding why our computing systems operate the way they do, the societal contexts in which these approaches arose, and how those continue to echo through the discipline today.

Topics:

[KA Core]

1. Age I: Prehistory—the world before ENIAC (1946): Ancient analog computing (Stonehenge, Antikythera mechanism, Salisbury Cathedral clock, etc.), Euclid, Lovelace, Babbage, Gödel, Church, Turing, pre-electronic (electro-mechanical and mechanical) hardware
2. Age II: Early modern (digital) computing - ENIAC, UNIVAC, Bombes (Bletchley Park codebreakers), mainframes, etc.
3. Age III: Modern (digital) computing - PCs, modern computer hardware, Moore's Law
4. Age IV: Internet - networking, internet architecture, browsers and their evolution, standards, big players (Google, Amazon, Microsoft, etc.), distributed computing
5. Age V: Cloud - smart phones (Apple, Android, and minor ones), cloud computing, remote servers, software as a service (SaaS), security and privacy, social media
6. Age VI: Emerging AI-assisted technologies including decision making systems, recommendation systems, generative AI and other machine learning driven tools and technologies

Illustrative Learning Outcomes:

[KA Core]

1. Identify significant trends in the history of the computing field.
2. Identify the contributions of several pioneers in the computing field.
3. Discuss the historical context for important moments in history of computing, such as: the move from vacuum tubes to transistors (TRADIC), the first real operating system (OS 360), Xerox PARC and the first Apple computer with a GUI, the creation of specific programming language paradigms, the first computer virus, the creation of the internet, the creation of the WWW, the dot com bust, Y2K, the introduction of smart phones, etc.
4. Compare daily life before and after the advent of personal computers and the Internet.

SEP/Economies of Computing

The economies of computing are important to those who develop and provide computing resources and services to others as well as society in general. They are equally important to users of these resources and services, both professional and non-professional.

Topics:

[KA Core]

1. Economies of providers: regulated and unregulated, monopolies and open-market. "Walled Gardens" in tech environments
2. The knowledge and attention economies
3. Effect of skilled labor supply and demand on the quality of computing products
4. Pricing strategies in the computing domain: subscriptions, planned obsolescence, software licenses, open-source, free software
5. Outsourcing and off-shoring software development; impacts on employment and on economics
6. Consequences of globalization for the computer science profession and users
7. Differences in access to computing resources and the possible effects thereof
8. Automation and its effect on job markets, developers, and users
9. Economies of scale, startups, entrepreneurship, philanthropy
10. How computing is changing personal finance: Blockchain and cryptocurrencies, mobile banking and payments, SMS payment in developing regions, etc.

Illustrative Learning Outcomes:

[KA Core]

1. Summarize concerns about monopolies in tech, walled gardens vs open environments, etc.
2. Identify several ways in which the information technology industry and users are affected by shortages in the labor supply.
3. Outline the evolution of pricing strategies for computing goods and services.
4. Explain the social effects of the knowledge and attention economies.
5. Summarize the consequences of globalization and nationalism in the computing industry.
6. Describe the effects of automation on society, and job markets in particular.
7. Detail how computing has changed the corporate landscape
8. Outline how computing has changed personal finance and the consequences of this, both positive and negative.

SEP/Security Policies, Laws and Computer Crimes

While security policies, laws and computer crimes are important, it is essential they are viewed with the foundation of other Social and Professional knowledge units, such as Intellectual Property, Privacy and Civil Liberties, Social Context, and Professional Ethics. Computers and the Internet, perhaps more than any other technology, have transformed society over the past 75 years. At the same time, they have contributed to unprecedented threats to privacy; new categories of computer crime and anti-social behavior; major disruptions to organizations; and the large-scale concentration of risk into information systems.

Topics:

[CS Core]

1. Examples of computer crimes and legal redress for computer criminals
2. Social engineering, computing-enabled fraud, and recovery
3. Identify what constitutes computer crime, such as Issues surrounding the misuse of access and breaches in security
4. Motivations and ramifications of cyber terrorism and criminal hacking, “cracking”
5. Effects of malware, such as viruses, worms and Trojans

[KA Core]

1. Crime prevention strategies
2. Security policies

Illustrative Learning Outcomes:

[CS Core]

1. List classic examples of computer crimes and social engineering incidents with societal impact.
2. Identify laws that apply to computer crimes.

3. Describe the motivation and ramifications of cyber terrorism and criminal hacking.
4. Examine the ethical and legal issues surrounding the misuse of access and various breaches in security.
5. Discuss the professional's role in security and the trade-offs involved.

[KA Core]

1. Investigate measures that can be taken by both individuals and organisations including governments to prevent or mitigate the undesirable effects of computer crimes and identity theft.
2. Write a company-wide security policy, which includes procedures for managing passwords and employee monitoring.

SEP/Diversity, Equity, and Inclusion

Computer Science has had—since its inception as a field—a diversity problem. Despite being a creative, highly compensated field with myriad job (and other) opportunities; racial and gender and other inequities in representation are pervasive. For too many students, their first computer science course is their last. There are many factors including the legacy of systemic racism, ableism, sexism, classism, and other injustices that contribute to the lack of diverse identities within computer science, and there is no single, quick fix.

CS2023's sponsoring organizations are ACM, IEEE CS, and AAAI. Each of those organizations [https://www.acm.org/diversity-inclusion/about#DEIPrinciples, https://www.ieee.org/about/diversity-index.html, https://aaai.org/Organization/diversity-statement.php] place a high value on diversity, equity and inclusion; and our computer science classrooms should promote and model those principles. We should welcome and seek diversity—the gamut of human differences including gender, gender identity, race, politics, ability and attributes, religion, nationality, etc.—in our classrooms, departments and campuses. We should strive to make our classrooms, labs and curricula accessible and to promote inclusion; the sense of belonging we feel in a community where we are respected and wanted. To achieve equity, we must allocate resources, promote fairness, and check our biases to ensure persons of all identities achieve success.

Explicitly infusing diversity, equity, and inclusion across the computer science curriculum demonstrates its importance for the department, institution, and our field—all of which likely have a DEI statement and/or initiative. This emphasis on DEI is important ethically and a bellwether issue of our time. Not only does data support that diverse teams outperform homogeneous ones, but diverse teams may have prevented egregious technology failures in the headlines such as facial recognition misuse, airbag injuries and deaths.

Topics:

[CS Core]

1. How identity impacts and is impacted by computing environments (academic and professional) and technologies

2. The benefits of diverse development teams and the impacts of teams that are not diverse.
3. Inclusive language and charged terminology, and why their use matters
4. Inclusive behaviors and why they matter
5. Technology and accessibility
6. How computing professionals, via the software they create, can influence and impact justice, diversity, equity, and inclusion both positively and negatively

[KA Core]

1. Highlight experts (practitioners, graduates, and upper level students) who reflect the identities of the classroom and the world
2. Benefits of diversity and harms caused by a lack of diversity
3. Historic marginalization due to technological supremacy and global infrastructure challenges to equity and accessibility

Illustrative Learning Outcomes:

[CS Core]

1. Define and distinguish equity, equality, diversity, and inclusion.
2. Describe the impact of power and privilege in the computing profession as it relates to culture, industry, products, and society.
3. What language, practices, and behaviors may make someone feel included in a workplace and/or a team, and why is it relevant. Avoiding charged terminology, see *Words Matter* (https://www.acm.org/diversity-inclusion/words-matter).
4. Evaluate the accessibility of your classroom or lab. Evaluate the accessibility of your webpage. (See https://www.w3.org/WAI/.)
5. Work collegially and respectfully with team members who do not share your identity. It is not enough to merely assign team projects. Faculty should prepare students for teamwork and monitor, mentor, and assess the effectiveness of their student teams throughout a project.
6. Compare the demographics of your institution's computer science and STEM majors to the overall institutional demographics. Do they differ? If so, what factors contribute to inequitable access, engagement, and achievement in computer science among marginalized groups.
7. Compare the demographics of your institution to the overall community demographics. Do they differ? If so, what factors contribute to inequitable access, engagement, and achievement among marginalized groups.
8. Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability by diverse populations.

[KA Core]

1. Highlight experts (practitioners, graduates, and upper level students—current and historic) who reflect the identities of the classroom and the world.
2. Identify examples of the benefits that diverse teams can bring to software products, and those where a lack of diversity have costs.

3. Give examples of systemic changes that could positively address diversity, equity, and inclusion in a familiar context (i.e. in an introductory computing course).

Professional Dispositions

- Professionalism - Relatively little in the practice of computing is entirely independent - we rely on others extensively, whether that is in the form of teamwork, customer relationships, leadership roles, academic research, etc. Being able to interact with others, regardless of identity, background, etc, is essential to success for both individuals and the industry as a whole.
- Responsibility - Responsibility, attention to detail, and awareness of potential social impact are highly desirable for computing graduates.
- Critical Self-reflection - Being able to inspect one's own actions, thoughts, biases, privileges, and motives will help in discovering places where professional activity is not up to current standards. Understand both conscious and unconscious bias and continuously work to counteract them.
- Responsiveness - Ability to quickly and accurately respond to changes in the field and adapt in a professional manner, such as shifting from in-person office work to remote work at home. These shifts require us to rethink our entire approach to what is considered "professional."
- Proactiveness - Being professional in the workplace means finding new trends (e.g. in accessibility or inclusion) and understanding how to implement them immediately for a more professional working environment.
- Cultural Competence - Prioritize cultural competence—the ability to work with people from cultures different from your own—by using inclusive language, watching for and counteracting conscious and unconscious bias, and encouraging honest and open communication.

Math Requirements

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Justice (SEP/Equity, Diversity, Equity and Inclusion) with all KAs
- Equity (SEP/Equity, Diversity, Equity and Inclusion) with all KAs
- Diversity (Equity, Diversity, Equity and Inclusion) with all KAs
- Inclusion (Equity, Diversity, Equity and Inclusion) with all KAs
- Social implications of computing in a hyper-networked world where the capabilities of artificial intelligence are rapidly evolving (SEP/Social Context) with
    - AI/
    - HCI/Foundations/social models

    - IAS/Fundamental Concepts/social issues)
- Growth and control of the Internet (SEP/Social Context) with NC/Introduction/organization of the Internet)
- Context-aware computing (SEP/Social Context) with HCI/Design for non-mouse interfaces/ ubiquitous and context-aware)
- Professional certification, codes of ethics, conduct, and practice, such as the ACM/IEEE-CS, SE, AITP, IFIP and international societies (SEP/Professional Ethics with IAS/Fundamental Concepts/ethical issues
- Intellectual property rights (SEP/Intellectual Property) with IM/Information Storage and Retrieval/intellectual property and protection
- Privacy implications of widespread data collection for transactional databases, data warehouses, surveillance systems, and cloud computing (SEP/Privacy and Civil Liberties) with
    - IM/Database Systems/data independence
    - IM/Data Mining/data cleaning
- Technology-based solutions for privacy protection (SEP/Privacy and Civil Liberties) with IAS/Threats and Attacks/attacks on privacy and anonymity
- Philosophical foundations of privacy rights (SEP/Privacy and Civil Liberties) with IS/Fundamental Issues/philosophical issues
- Identifying, describing, and employing (clear, polite, concise) oral, written, and electronic team and group communication (SEP/Professional Communication) with
    - HCI/Collaboration and Communication/group communication
    - SE/Project Management/team participation
- Utilizing collaboration tools (SEP/Professional Communication) with HCI/Collaboration and Communication/online communities)
- Understanding accessibility (SEP/Professional Communication) with HCI/*
- Demonstrate cultural competence in written and verbal communication (SEP/Professional Communication) with
    - HCI/User-Centered Design and Testing/cross-cultural evaluation
- Using synthesis to concisely and accurately convey tradeoffs in competing values driving software projects including technology, structure/process, quality, people, market and financial (SEP/Professional Communication) with SE/Software Project Management/Risk
- Age III: Modern (digital) computing - PCs, modern computer hardware (SEP/History) with AR/Digital logic and digital systems/ history of computer architecture
- Examples of computer crimes and legal redress for computer criminals (SEP/Security Policies, Laws and Computer Crimes) with IAS/Digital Forensics/rules of evidence
- Social engineering, computing-enabled fraud, and recovery (SEP/Security Policies, Laws and Computer Crimes) with HCI/Human Factors and Security/trust, privacy and deception
- Security policies (SEP/Security Policies, Laws and Computer Crimes) with IAS/Security Policy and Governance/policies

Crosscutting themes:

- Justice
- Equity
- Diversity
- Inclusion
- Societal Issues
- Ethics
- Professionalism

Course Packaging Suggestions

At a minimum the CS Core learning outcomes are best covered in the context of courses covering other knowledge areas. Ideally the KA Core hours are also.

At some institutions a stand-alone course (possibly at a mid-level) may be offered covering the CS Core knowledge units:

- Social Context: 3 hours
- Methods for Ethical Analysis: 2 hours
- Professional Ethics: 2 hours
- Intellectual Property: 1 hour
- Privacy and Civil Liberties: 2 hours
- Professional Communication: 2 hours
- Sustainability: 1 hour
- Security Policies, Laws and Computer Crimes: 2 hours
- Justice. Equity, Diversity and Inclusion: 2 hours

At some institutions a stand-alone course at the mid- or advanced-level may be offered covering the CS Core and KA Core knowledge units:

- Social Context: 5 hours
- Methods for Ethical Analysis: 3 hours
- Professional Ethics: 4 hours
- Intellectual Property: 2 hours
- Privacy and Civil Liberties: 3 hours
- Professional Communication: 3 hours
- Sustainability: 2 hours
- History: 1 hour
- Economies of Computing: 1 hour
- Security Policies, Laws and Computer Crimes: 3 hours
- Justice. Equity, Diversity and Inclusion: 4 hours

Competency Specifications

- **Task 1:** Assess the ethical and societal implications of deploying a given AI-powered service/software/product
- **Competency Statement:** Determine who will be affected and how
- **Competency area:** Theory
- **Competency unit:** Deployment / Evaluation / Consumer Acceptance / Adaptation to social issues
- **Required knowledge areas and knowledge units:**
    - AI/Fundamental Issues
    - SEP/Social Context
    - SEP/Methods for Ethical Analysis
    - SEP/Privacy and Civil Liberties
    - SEP/Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain / Evaluate
- **Core level:** CS core

- **Task 2:** Assess the legal and ethical implications of collecting and using customer/user data
- **Competency Statement:** Analyze the nature of the data and determine the relevant legal and ethical issues that may arise in the collection and use of that data
- **Competency area:** Theory
- **Competency unit:** Evaluation / Consumer Acceptance / Adaptation to social issues
- **Required knowledge areas and knowledge units:**
    - DM/Data Security & Privacy
    - SEP/Methods for Ethical Analysis
    - SEP/Privacy and Civil Liberties
    - SEP/Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain / Evaluate
- **Core level:** CS core

- **Task 3:** Evaluate the role culture plays in who adopts a given service/software/product, how they may use it, and how this impacts equity and society
- **Competency Statement:** Apply the appropriate methods of ethical analysis to determine the social context(s) applicable and the potential ethical effects
- **Competency area:** Theory
- **Competency unit:** Deployment / Evaluation / Adaptation to social issues

- **Required knowledge areas and knowledge units:**
    - SEP/Social Context
    - SEP/Methods for Ethical Analysis
    - SEP/Privacy and Civil Liberties
    - SEP/Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain / Evaluate
- **Core level:** CS core

- **Task 4:** Document the accountability, responsibility and liability a company assumes when releasing a given service/software/product
- **Competency Statement:** Gather the appropriate information and present it in a coherent and useful manner
- **Competency area:** Application / Theory
- **Competency unit:** Requirements / Deployment / Documentation
- **Required knowledge areas and knowledge units:**
    - SEP/Professional Ethics
    - SEP/Intellectual Property
    - SEP/Privacy and Civil Liberties
    - SEP/Professional Communication
    - SEP/Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain
- **Core level:** CS core

- **Task 5:** Develop a strategy for a team to keep up to date with ethical, legal, and professional issues in relation to company strategy
- **Competency Statement:** Apply the ability to assess company strategy and formulate a plan to communicate the relevant information to a team through appropriate professional documentation and communication
- **Competency area:** Application / Theory
- **Competency unit:** Requirements / Design / Development / Documentation / Evaluation / Management / Adaptation to social issues / Improvement
- **Required knowledge areas and knowledge units:**
    - SEP/Professional Communication
    - SEP/Professional Ethics
- **Required skill level:** Explain / Evaluate / Develop

- **Core level:** CS core

- **Task 6:** Incorporate legal and ethical privacy requirements into a given service/software/product's development cycle
- **Competency Statement:** Identify and analyze appropriate requirements and develop a plan to communicate a plan to ensure the requirements are implemented
- **Competency area:** Software / Theory
- **Competency unit:** Requirements / Design / Development / Documentation Management / Improvement
- **Required knowledge areas and knowledge units:**
    - SEP/Professional Ethics
    - SEP/Privacy and Civil Liberties
    - SEP/Professional Communication
- **Required skill level:** Explain / Develop
- **Core level:** CS core

- **Task 7:** Demonstrate cultural awareness and cultural competence in written and verbal communication
- **Competency Statement:** Professionally communicate with required stakeholders with cultural sensitivity
- **Competency area:** Theory
- **Competency unit:** Documentation / Adaptation to social issues
- **Required knowledge areas and knowledge units:**
    - SEP/Social Context
    - SEP/Professional Communication
    - SEP/Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain
- **Core level:** CS core

- **Task 8:** Contribute to a company or team's sustainability policy
- **Competency Statement:** Effectively integrate knowledge of sustainability with company or team strategy and mission
- **Competency area:** Theory
- **Competency unit:** Design / Development / Integration / Documentation / Improvement
- **Required knowledge areas and knowledge units:**
    - SEP/Professional Communication
    - SEP/Sustainability
- **Required skill level:** Explain / Evaluate / Develop
- **Core level:** CS core

- **Task 9:** Convey the benefits of diverse development teams and user bases on company culture and the services/software/products the company provides, as well as the impacts that a lack of diversity can have on these
- **Competency Statement:** Professionally communicate knowledge of the impacts of diversity
- **Competency area:** Theory
- **Competency unit:** Documentation / Management / Adaptation to social issues / Improvement
- **Required knowledge areas and knowledge units:**
    - SEP/Professional Communication
    - SEP/Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain
- **Core level:** CS core

- **Task 10:** Contribute to the design and implementation of systemic changes that could positively address diversity, equity, and inclusion in the company's employees and the user base
- **Competency Statement:** Apply knowledge of diversity, equity, and inclusion in an effective manner in the given context
- **Competency area:** Application / Theory
- **Competency unit:** Requirements / Design / Consumer Acceptance / Adaptation to social issues / Improvement
- **Required knowledge areas and knowledge units:**
    - SEP/Professional Communication
    - SEP/Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain / Apply / Evaluate / Develop
- **Core level:** CS core

Committee

Chair: Brett A. Becker, University College Dublin, Ireland

Members:

- Richard L. Blumenthal, Regis University, Denver, CO, USA
- Michael Goldweber, Xavier University, Cincinnati, OH, USA
- James Prather, Abilene Christian University, TX, USA
- Susan Reiser, University of North Carolina Asheville, NC, USA
- Michelle Trim, University of Massachusetts Amherst, MA, USA
- Titus Winters, Google, Inc, New York, NY, USA

Contributors:

- Johanna Blumenthal, Regis University, Denver, CO, USA
- MaryAnne Egan, Siena College, NY, USA
- Keith Quille, Technological University of Dublin, Dublin, Ireland
- Mehran Sahami, Stanford University, CA, USA
- Mark Scanlon, University College Dublin, Dublin, Ireland
- Karren Shorofsky, University of San Francisco School of Law, CA, USA
- Ellen Walker, Hiram College, OH, USA

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
SEP	Social Context	1. Social implications of computing in a hyper-networked world where the capabilities of artificial intelligence are rapidly evolving 2. Impact of social media and artificial intelligence on individual well-being, political ideology, and cultural ideology 3. Impact of involving computing technologies, particularly artificial intelligence, biometric technologies and algorithmic decision-making systems, in civic life (e.g. facial recognition technology, biometric tags, resource distribution algorithms, policing software)	Evaluate	CS	3
SEP	Social Context	1. Growth and control of the internet, computing, and artificial intelligence 2. Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or developing nations 3. Accessibility issues, including legal requirements and dark patterns 4. Context-aware computing	Explain	KA	2
SEP	Methods for Ethical Analysis	1. Avoiding fallacies and misrepresentation in argumentation 2. Ethical theories and decision-making (philosophical and social frameworks and epistemologies) 3. Recognition of the role culture plays in our understanding, adoption, design, and use of computing technology	Apply	CS	2
SEP	Methods for Ethical Analysis	1. Professional checklists 2. Evaluation rubrics 3. Stakeholder analysis 4. Standpoint theory	Develop	KA	1
SEP	Professional Ethics	1. Community values and the laws by which we live 2. The nature of professionalism including care, attention and discipline, fiduciary responsibility, and mentoring	Evaluate	CS	2

		3. Keeping up-to-date as a computing professional in terms of familiarity, tools, skills, legal and professional frameworks as well as the ability to self-assess and progress in the computing field 4. Professional certification, codes of ethics, conduct, and practice, such as the ACM/IEEE-CS, SE, AITP, IFIP and international societies 5. Accountability, responsibility and liability (e.g. software correctness, reliability and safety, as well as ethical confidentiality of cybersecurity professionals) 6. Introduction to theories describing the human creation and use of technology including instrumentalism, sociology of technological systems, disability justice, neutrality thesis, pragmatism, utilitarianism, and decolonial theories 7. Develop strategies for recognizing and reporting designs, systems, software, and professional conduct (or their outcomes) that may violate law or professional codes of ethics			
SEP	Professional Ethics	1. The role of the computing professional in public policy 2. Maintaining awareness of consequences 3. Ethical dissent and whistle-blowing 4. The relationship between regional culture and ethical dilemmas 5. Dealing with harassment and discrimination 6. Forms of professional credentialing 7. Acceptable use policies for computing in the workplace 8. Ergonomics and healthy computing environments 9. Time to market and cost considerations versus quality professional standards	Explain	KA	2
SEP	Intellectual Property	1. Philosophical foundations of intellectual property 2. Intellectual property rights 3. Intangible digital intellectual property (IDIP) 4. Legal foundations for intellectual property protection	Explain	CS	1

SEP	Intellectual Property	1. Digital rights management 2. Copyrights, patents, trade secrets, trademarks 3. Plagiarism 4. Foundations of the open source movement 5. Software piracy	Apply	KA	1
SEP	Privacy and Civil Liberties	1. Privacy implications of widespread data collection for transactional databases, data warehouses, surveillance systems, and cloud computing 2. Ramifications of differential privacy 3. Technology-based solutions for privacy protection 4. Civil liberties and cultural differences	Explain	CS	2
SEP	Privacy and Civil Liberties	1. Philosophical foundations of privacy rights 2. Legal foundations of privacy protection in relevant jurisdictions 3. Privacy legislation in areas of practice 4. Freedom of expression and its limitations	Assess	KA	1
SEP	Professional Communication	1. Interpreting, summarising, and synthesising technical material, including source code and documentation 2. Writing effective technical documentation and materials (tutorials, reference materials, API documentation) 3. Identifying, describing, and employing (clear, polite, concise) oral, written, and electronic team and group communication. 4. Understanding and enacting awareness of audience in communication by communicating effectively with different customers, stakeholders, and leadership 5. Utilising collaboration tools 6. Recognizing and avoiding the use of rhetorical fallacies when resolving technical disputes 7. Understanding accessibility and inclusivity requirements for addressing professional audiences	Apply	CS	2
SEP	Professional	1. Demonstrate cultural competence in written and verbal communication 2. Using synthesis to concisely and accurately convey tradeoffs in competing values driving	Apply	KA	1

	Communication	software projects including technology, structure/process, quality, people, market and financial 3. Use writing to solve problems or make recommendations in the workplace, such as raising ethical concerns or addressing accessibility issues			
SEP	Sustainability	1. Being a sustainable practitioner by taking into consideration environmental, social, and cultural impacts of implementation decisions (e.g. algorithmic bias/outcomes, , economic viability, and resource consumption) 2. Explore local/regional/global social and environmental impacts of computing systems use and disposal (e-waste) 3. Discuss the tradeoffs involved in proof-of-work and proof-of-stake algorithm	Apply	CS	1
SEP	Sustainability	1. Guidelines for sustainable design standards 2. Systemic effects of complex computer-mediated phenomena (e.g. social media, offshoring, remote work) 3. Pervasive computing: Information processing that has been integrated into everyday objects and activities, such as smart energy systems, social networking and feedback systems to promote sustainable behavior, transportation, environmental monitoring, citizen science and activism 4. Conduct research on applications of computing to environmental issues, such as energy, pollution, resource usage, recycling and reuse, food management / production, and others 5. How the sustainability of software systems are interdependent with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g. market forces, government policies)	Evaluate	KA	1
SEP	History	1. Age I: Prehistory—the world before ENIAC (1946): Ancient analog computing (Stonehenge, Antikythera mechanism, Salisbury Cathedral clock, etc.), Euclid, Lovelace, Babbage, Gödel, Church, Turing,	Explain	KA	1

		pre-electronic (electro-mechanical and mechanical) hardware 2. Age II: Early modern (digital) computing - ENIAC, UNIVAC, Bombes (Bletchley Park codebreakers), mainframes, etc. 3. Age III: Modern (digital) computing - PCs, modern computer hardware, Moore's Law 4. Age IV: Internet - networking, internet architecture, browsers and their evolution, standards, big players (Google, Amazon, Microsoft, etc.), distributed computing 5. Age V: Cloud - smart phones (Apple, Android, and minor ones), cloud computing, remote servers, software as a service (SaaS), security and privacy, social media 6. Age VI: Emerging AI-assisted technologies including decision making systems, recommendation systems, generative AI and other machine learning driven tools and technologies			
SEP	Economies of Computing	1. Economies of providers: regulated and unregulated, monopolies and open-market. "Walled Gardens" in tech environments 2. The knowledge and attention economies 3. Effect of skilled labor supply and demand on the quality of computing products 4. Pricing strategies in the computing domain: subscriptions, planned obsolescence, software licenses, open-source, free software 5. Outsourcing and off-shoring software development; impacts on employment and on economics 6. Consequences of globalization for the computer science profession and users 7. Differences in access to computing resources and the possible effects thereof 8. Automation and its effect on job markets, developers, and users 9. Economies of scale, startups, entrepreneurship, philanthropy 10. How computing is changing personal finance: Blockchain and cryptocurrencies, mobile banking and payments, SMS payment in developing regions, etc.	Explain	KA	1

SEP	Security Policies, Laws and Compute r Crimes	1. Examples of computer crimes and legal redress for computer criminals 2. Social engineering, computing-enabled fraud, and recovery 3. Identify what constitutes computer crime, such as Issues surrounding the misuse of access and breaches in security 4. Motivations and ramifications of cyber terrorism and criminal hacking, “cracking” 5. Effects of malware, such as viruses, worms and Trojans	Explain	CS	2
SEP	Security Policies, Laws and Compute r Crimes	1. Crime prevention strategies 2. Security policies	Apply	KA	1
SEP	Equity, Diversity and Inclusion	1. How identity impacts and is impacted by computing environments (academic and professional) and technologies 2. The benefits of diverse development teams and the impacts of teams that are not diverse. 3. Inclusive language and charged terminology, and why their use matters 4. Inclusive behaviors and why they matter 5. Technology and accessibility 6. How computing professionals, via the software they create, can influence and impact justice, diversity, equity, and inclusion both positively and negatively	Explain	CS	2
SEP	Equity, Diversity and Inclusion	1. Highlight experts (practitioners, graduates, and upper level students) who reflect the identities of the classroom and the world 2. Benefits of diversity and harms caused by a lack of diversity 3. Historic marginalization due to technological supremacy and global infrastructure challenges to equity and accessibility	Evaluate	KA	2

Software Development Fundamentals (SDF)

Preamble

Fluency in the process of software development is fundamental to the study of computer science. In order to use computers to solve problems most effectively, students must be competent at reading and writing programs in multiple programming languages. Beyond programming skills, however, they must be able to select and use appropriate data structures and algorithms, and utilize modern development and testing tools.

The SDF knowledge area brings together fundamental concepts and skills related to software development, focusing on concepts and skills that should be mastered early in a computer science program, typically in the first year. This includes fundamental programming concepts and their effective use in writing programs, use of fundamental data structures which may be provided by the programming language, basics of programming practices for writing good quality programs, and some understanding of the impact of algorithms on the performance of the programs. The 43 hours of material in this knowledge area may be augmented with core material from other knowledge areas as students progress to mid- and upper-level courses.

This knowledge area assumes a contemporary programming language with good built-in support for common data types including associative data types like dictionaries/maps as the vehicle for introducing students to programming (e.g. Python, Java). However, this is not to discourage the use of older or lower-level languages for SDF - the knowledge units below can be suitably adapted for the actual language used. The main change from 2013 is a stronger emphasis on developing fundamental programming skills and effective use of in-built data structures (which many contemporary languages provide) for problem solving.

Changes since CS 2013:

Core Hours

Knowledge Units	CS Core	KA Core
Fundamental Programming Concepts and Practices	20	
Fundamental Data Structures	12	
Algorithms	6	

Software Development Practices	5	
Total	**43**	

Knowledge Units

SDF/Fundamental Programming Concepts and Practices

[20 Core-Tier1 hours]

This knowledge unit aims to develop core programming concepts through one or more programming languages. It focuses on understanding of basic concepts (e.g., variables, types, expressions), and fluent use of basic constructs (e.g., assignments, conditionals, iteration) as well as modularity constructs (e.g., functions, classes/objects). It also aims to familiarize students with the concept of common libraries and frameworks, including those to facilitate API-based access to resources.

Topics

- Basic concepts such as variables, primitive data types, expression evaluation, assignment, etc.
- Basic constructs such as conditional and iterative structures and flow of control
- Key modularity constructs such as functions (and methods and classes, if supported in the language) and related concepts like parameter passing, scope, abstraction, data encapsulation, etc.
- Input and output using files, console, and APIs
- Structured data types available in the chosen programming language like sequences (e.g., arrays, lists), associative containers (e.g., dictionaries, maps), others (e.g., sets, tuples) and when and how to use them.
- Libraries and frameworks provided by the language (when/where applicable)
- Recursion
- Dealing with runtime errors in programs (exception handling)
- Basic concept of programming errors, testing, and debugging
- Reading and understanding code

Illustrative Learning Outcomes

1. Design, code, test, and debug a program that uses each of the following fundamental programming constructs: assignment and expressions, simple I/O, conditional and iterative structures, functions with parameter passing.
2. Design, code, test, and debug a program that effectively uses the different structured data types provided in the language like strings, arrays/lists, dictionaries, sets
3. Write a program that uses file I/O to provide persistence across multiple executions.
4. Write a program that uses APIs to get data (e.g., from the web, where applicable)
5. Write a program that uses some language-provided libraries and frameworks (where applicable).
6. Write a program that creates simple classes and instantiates objects of those classes (if supported by the language)
7. Explain the concept of recursion, and identify when and how to use it effectively
8. Write recursive functions
9. Write a program that can handle a runtime exception.
10. Read and interpret code segments provided
11. Trace the flow of control during the execution of a program (both correct and incorrect).
12. Use appropriate terminology to identity elements of a program (e.g., identifier, operator, operand)

SDF/Fundamental Data Structures

[12 Core-Tier1 hours]

This unit aims to develop core concepts relating to Data Structures including associated algorithms. Students should understand the important data structures (often available in the programming language or as libraries) for modern applications, and how to use them effectively. This includes choosing appropriate data structures while designing solutions for a given problem.

Topics

- Standard abstract data types such as lists, stacks, queues, sets, and maps/dictionaries
- When and how to use standard data structures
- Strings and string processing
- Performance implications of choice of data structure(s)

Illustrative Learning Outcomes

- Write programs that use each of the key abstract data types / data structures provided in the language (e.g., arrays, tuples/records/structs , lists, stacks, queues, and associative data types like sets, dictionaries/maps.
- Select the appropriate data structure for a given problem.
- Write programs that work with text by using string processing capabilities provided by the language.

- Measure the performance of a program (e.g. to assess how performance changes with scale, alternative data structures, …).

SDF/Algorithms

[6 Core-Tier1 hours]

This unit aims to develop the foundations of efficient algorithms and their analysis. The KU should also empower students in selecting suitable algorithms for building modest-complexity applications.

Topics

- Concept of algorithm and notion of algorithm efficiency
- Common algorithms like: Sorting, Searching, Tree traversal, Graph traversal, etc.
- Assessing the time/space efficiency of algorithms through measurement

Illustrative Learning Outcomes

- Explain the importance of algorithms in the problem-solving process.
- Demonstrate how a problem may be solved by multiple algorithms, each with different properties.
- Describe common algorithms like: Sorting, Searching, Tree traversal, Graph traversal, etc.
- Experiment with space/time performance of some algorithms.

SDF/Software Development Practices

[5 Core-Tier1 hours]

This unit develops the core concepts relating to modern software development practices. Its aim is to develop student understanding and basic competencies in using modern methods and tools, including some general purpose IDE, use of debuggers, testing, etc.

Topics:

- Basic testing (perhaps using suitable frameworks) including test case design
- Use of a general purpose IDE, including its debugger (which can be also used to strengthen some programming concepts)
- Programming style that improves readability

Illustrative Learning Outcomes

- Apply basic programming style guidelines to aid readability of programs such as comments, indentation, proper naming of variables, etc.
- Build, execute and debug programs using a modern IDE and associated tools such as visual debuggers.

- Develop tests for modules, and apply a variety of strategies to design test cases (perhaps using a testing framework).
- Explain some limitations of testing programs

Professional Dispositions

- Self-Directed. Seeking out solutions to issues on their own (e.g., using technical forums, FAQs, discussions).
- Experimental. Practical experimentation characterized by experimenting with language features to understand them, quickly prototyping approaches, using the debugger to understand why a bug is occurring, etc. .
- Technical curiosity. Characterized by, for example, interest in understanding how programs are executed, what is happening in IDE/editor, how programs and data are stored, etc.
- Technical adaptability. Characterized by willingness to learn about and use different tools and technologies that facilitate software development.
- Perseverance. To continue efforts till, for example, a bug is identified, a program is robust and handles most of the situations, etc.
- Systematic. Characterized by attention to detail and use of orderly processes in practice.

Math Requirements

As SDF focuses on the first year and is foundational, it assumes only basic math knowledge that students acquire in school.

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Software Engineering (SE). All the topics/LOs mentioned under the KA "Software Development Practices"
- Algorithms and Complexity (AL): All topics/LOs listed under the KA "Algorithms"

Course Packaging Suggestions

The SDF KA will generally be covered in introductory courses, often called CS1 and CS2. How much of the SDF KA can be covered in CS1 and how much is to be left for CS2 is likely to depend on the choice of

programming language for CS1. For languages like Python or Java, CS1 can cover all of the Programming Concepts and Development Methods KAs, and some of the Data Structures KA. It is desirable that they be further strengthened in CS2. The topics under algorithms KA and some topics under data structures KA can be covered in CS2. In case CS1 uses a language with fewer in-built data structures, then much of the Data Structures KA and some aspects of the programming KA may also need to be covered in CS2. With the former approach, the introductory course in programming can include the following:

1. SDF/Fundamental Programming Concepts and Practices, 20 hours
2. SDF/Fundamental Data Structures, 12 hours
3. SDF/Algorithms, 2 hours (the remaining 4 hours should be covered in CS2 via AL)
4. SDF/Software Development Practices, 2 hours (the remaining 3 hrs should be covered via SE)
5. KUs from SEP Knowledge Area: 2 to 4 hours (exact KUs to be discussed with SEP)

Pre-requisites: School Maths (Sets, Relations, and Logic)

Skill statement: A student who completes this course should be able to:

- Design, code, test, and debug a modest sized program that effectively uses the functional abstraction.
- Select and use the appropriate language provided data structure for a given problem (like: arrays, tuples/records/structs , lists, stacks, queues, and associative data types like sets, dictionaries/maps.)
- Design, code, test, and debug a modest-sized object oriented program using classes and objects.
- Design, code, test, and debug a modest-sized program that uses language provided libraries and frameworks (including for getting data from the web through APIs)
- Read and interpret given code including tracing the flow of control during execution
- Build, execute and debug programs using a modern IDE and associated tools such as visual debuggers.
- Explain the key concepts relating to programming like parameter passing, recursion, runtime exceptions and exception handling.

Competency Specifications

Given that SDF is for entry level programming capability, the term program in these means a modest size program.

- **Task 1:** Given specifications, develop a program to implement it.
- **Competency Statement:** Write code for a function/class or a small program.
- **Competency area:** Software

- **Competency unit:** Development, Testing
- **Required knowledge areas and knowledge units:**
    - SDF/ Fundamental Programming Concepts
- **Required skill level:** Develop
- **Core level:** CS core

- **Task 2:** Write documentation for a program.
- **Competency Statement:** Read and understand given code and explain it.
- **Competency area:** Software
- **Competency unit:** Documentation
- **Required knowledge areas and knowledge units:**
    - SDF/ Fundamental Programming Concepts
- **Required skill level:** Explain
- **Core level:** CS core

- **Task 3:** Develop test cases to determine if a program is functionally correct.
- **Competency Statement:** Develop test cases and test a given program.
- **Competency area:** Software
- **Competency unit:** Testing
- **Required knowledge areas and knowledge units:**
    - SDF/ Software Development Practices
- **Required skill level:** Develop
- **Core level:** CS core

- **Task 4:** Identify and fix a bug in a program.
- **Competency Statement:** Debug a program.
- **Competency area:** Software
- **Competency unit:** Development, Testing, Evaluation, Maintenance

- **Required knowledge areas and knowledge units:**
    - SDF/ Fundamental Programming Concepts, Software Development Practices
- **Required skill level:** Evaluate / Develop
- **Core level:** CS core

- **Task 5:** Perform a code review to evaluate the quality of code.
- **Competency Statement:** Read and understand the code and identify errors in it.
- **Competency area:** Software, Application
- **Competency unit:** Documentation, Evaluation, Improvement
- **Required knowledge areas and knowledge units:**
    - SDF/ Fundamental Programming Concepts, Software Development Practices
- **Required skill level:** Evaluate
- **Core level:** CS core

- **Task 6:** Explain a program at different levels of abstraction.
- **Competency Statement:** Read and understand code.
- **Competency area:** Software
- **Competency unit:** Documentation / Evaluation
- **Required knowledge areas and knowledge units:**
    - SDF/ Fundamental Programming Concepts
- **Required skill level:** Explain, Evaluate
- **Core level:** CS core

- **Task 7:** For a given programming problem, select the most appropriate data structure.
- **Competency Statement:** Use appropriate data structures to write code for solving a problem.
- **Competency area:** Software, Application

- **Competency unit:** Development, Evaluation
- **Required knowledge areas and knowledge units:**
    - SDF/ Fundamental Programming Concepts, Fundamental Data Structures
- **Required skill level:** Evaluate / Develop
- **Core level:** CS core

- **Task 8:** Develop a program that effectively leverages the capabilities of libraries and APIs.
- **Competency Statement:** Write a program using APIs and/or Libraries.
- **Competency area:** Software, Application
- **Competency unit:** Development
- **Required knowledge areas and knowledge units:**
    - SDF/ Fundamental Programming Concepts
- **Required skill level:** Develop
- **Core level:** CS core

Committee

Chair: Pankaj Jalote, Chair, IIIT-Delhi, Delhi, India

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Titus Winters, Google, New York City, NY, USA
- Andrew Luxton-Reilly, University of Auckland, New Zealand
- Viraj Kumar, ACM India Education Committee, India
- Christian Servin, El Paso Community College, El Paso, TX, USA
- Karen Reid, University of Toronto, Toronto, Canada
- Adrienne Decker, University at Buffalo, Buffalo, USA

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
SDF	Fundamental Programming Concepts	● Basic concepts such as variables, primitive data types, expression evaluation, assignment, etc. ● Basic constructs such as conditional and iterative structures and flow of control ● Key modularity constructs such as functions/methods and classes, and related concepts like parameter passing, scope, abstraction, data encapsulation, etc. ● Input and output using files, console, and APIs ● Structured data types available in the chosen programming language like sequences ● Libraries and frameworks provided by the language (when/where applicable) ● Recursion	Develop	CS	18
SDF	Fundamental Programming Concepts	● Basic concept of programming errors, testing, and debuggingDealing with compile time and runtime errors ● Reading and understanding code	Evaluate, Apply	CS	2
SDF	Fundamental Data Structures	● Standard abstract data types such as lists, stacks, queues, sets, and maps/dictionaries [Shared with: AL] ● Strings and string processing	Develop	CS	10
SDF	Fundamental Data Structures	● When and how to use standard data structures ● Performance implications of choice of data structure(s)	Evaluate	CS	2
SDF	Algorithms	● Concept of algorithm and notion of algorithm efficiency ● Common algorithms like: Sorting, Searching,	Explain	CS	4

		Tree traversal, Graph traversal, etc. [Shared with AL]			
SDF	Algorithms	● Assessing the time/space efficiency of a algorithms through measurement [Shared with AL]	Evaluate	CS	2
SDF	Software Development practices	● Programming style that improves readability [Shared with SE]	Evaluate	CS	1
SDF	Software Development practices	● Basic unit testing (using suitable frameworks) including test case design [Shared with SE]	Develop	CS	2
SDF	Software Development practices	● Use of a general purpose IDE, including its debugger (which can be also used to strengthen some programming concepts)	Apply	CS	2

Software Engineering (SE)

Preamble

As far back as the early 1970s, Dave Parnas allegedly said, “Software engineering is the multi-person construction of multi-version programs.” This is an essential insight: while programming is the skill that governs our ability to write a program, software engineering is distinct in two dimensions: time and people.

First, a software engineering project is a team endeavor. Being a solitary programming expert is insufficient. Skilled software engineers will additionally demonstrate expertise in communication and collaboration. Programming may be an individual activity, but software engineering is a collaborative one, deeply tied to issues of professionalism, teamwork, and communication.

Second, a software engineering project is usually “multi-version.” It has an expected lifespan; it needs to function properly for months, years, or decades. Features may be added or removed to meet product requirements. The technological context will change, as our computing platforms evolve, programming languages change, dependencies upgrade, etc. This exposure to matters of time and change is novel when compared to a programming project: it isn’t enough to build a thing that works, instead it must work and stay working. Many of the most challenging topics in tech share “time will lead to change” as a root cause: backward compatibility, version skew, dependency management, schema changes, protocol evolution.

Software engineering presents a particularly difficult challenge for learning in an academic setting. Given that the major differences between programming and Software engineering are time and teamwork, it is hard to generate lessons that *require* successful teamwork and that faithfully present the risks of time. Additionally, some topics in software engineering will be more authentic and more relevant if and when our learners experience collaborative and long-term software engineering projects *in vivo* rather than in the classroom. Regardless of whether that happens as an internship, involvement in an open source project, or full-time engineering role, a month of full-time hands-on experience has more available hours than the average software engineering course.

Thus, a software engineering curriculum should focus primarily on ideas that are needed by a majority of new-grad hires, and that either are novel for those who are trained primarily as programmers, or that are abstract concepts that may not get explicitly stated/shared on the job. Such topics include, but are not limited to:

- Testing
- Teamwork, collaboration
- Communication
- Design
- Maintenance and Evolution
- Software engineering tools

Some such material is reasonably suited to a standard lecture or lecture+lab course. Discussing theoretical underpinnings of version control systems, or branching strategies in such systems, can be an effective way to familiarize students with those ideas. Similarly, a theoretical discussion can highlight the difference between static and dynamic analysis tools, or may motivate discussion of diamond dependency problems in dependency networks.

On the other hand, many of the fundamental topics of software engineering are best experienced in a hands-on fashion. Historically, project-oriented courses have been a common vehicle for such learning. We believe that such experience is valuable but also bears some interesting risks: students may form erroneous notions about the difficulty / complexity of collaboration if their only exposure is a single project with teams formed of other novice software engineers. It falls to instructors to decide on the right balance between theoretical material and hands-on projects - neither is a perfect vehicle for this challenging material. We strongly encourage instructors of project courses to aim for iteration and fast feedback - a few simple tasks repeated (i.e. in an Agile-structured project) is better than singular high-friction introductions to many types of tasks. If long-running project courses are not an option, anything that can expose learners to the collaborative and long-term aspects of software engineering is valuable: adding features to an existing codebase, collaborating on distinct parts of a larger whole, pairing up to write an encoder and decoder, etc.

All evidence suggests that the role of software in our society will continue to grow for the foreseeable future, and yet the era of “two programmers in a garage” seems to have drawn to a close. Most important software these days is clearly a team effort, building on existing code and leveraging existing functionality. The study of software engineering skills is a deeply important counterpoint to the everyday experience of computing students - we *must* impress on them the reality that few software projects are managed by writing from scratch as a solo endeavor. Communication, teamwork, planning, testing, and tooling are far more important as our students move on from the classroom and make their mark on the wider world.

Changes since CS 2013: This document shifts the focus of the Software Engineering knowledge area in a few ways compared to the goals of CS2013. The common reasoning behind most of these changes is to focus on material that learners would not pick up elsewhere in the curriculum, and that will be relevant *immediately* upon graduation, rather than at some future point in their careers.

- More explicit focus on the software workflow (version control, testing, code review, tooling)
- Less focus on team *leadership* and project management.
- More focus on team *participation*, communication, and collaboration

Core Hours

Knowledge Units	CS Core	KA Core
Teamwork	2	2
Tools and Environments	1	3
Product Requirements		2
Software Design	1	4
Software Construction	1	3
Software Verification and Validation	1	3
Refactoring and Code Evolution		2
Software Reliability		2
Formal Methods		
Total	**6**	**21**

Knowledge Units

SE/Teamwork

[2 CS Core hours, 2 KA Core hours]

Because of the nature of learning programming, most students in introductory SE have little or no exposure to the collaborative nature of SE. Practice (for instance in project work) may help, but lecture and discussion time spent on the value of clear, effective, and efficient communication and collaboration. are essential for Software Engineering.

Topics:

[CS Core]

- Effective communication
- Common causes of team conflict, and approaches for conflict resolution
- Cooperative programming
    - Pair programming
    - Code review
- Roles and responsibilities in a software team
    - Advantages of teamwork
    - Risks and complexity of such collaboration
- Team processes
    - Responsibilities for tasks, effort estimation, meeting structure, work schedule
- Importance of team diversity

[KA Core]

- Interfacing with stakeholders, as a team
    - Management & other non-technical teams
    - Customers
    - Users
- Risks associated with physical, distributed, hybrid and virtual teams
    - Including communication, perception, structure, points of failure, mitigation and recovery, etc.

Learning Outcomes:

[CS Core]

1. Follow effective team communication practices.
2. Articulate the sources of, hazards of, and potential benefits of team conflict - especially focusing on the value of disagreeing about ideas or proposals without insulting people.
3. Facilitate a conflict resolution strategy in a team setting.
4. Collaborate effectively in cooperative development/programming.
5. Propose and delegate necessary roles and responsibilities in a software development team.
6. Compose and follow an agenda for a team meeting.
7. Facilitate through involvement in a team project, the central elements of team building, establishing healthy team culture, and team management including creating and executing a team work plan.
8. Promote the importance of and benefits that diversity brings to a software development team

[KA Core]

9. Reference the importance of, and strategies to, as a team, interface with stakeholders not on the team on both technical and non-technical levels.
10. Enumerate the risks associated with physical, distributed, hybrid and virtual teams and possible points of failure and how to mitigate against and recover/learn from failures.

SE/Tools and Environments

Industry reliance on SE tools has exploded in the past generation, with version control becoming ubiquitous, testing frameworks growing in popularity, increased reliance on static and dynamic analysis in practice, and near-ubiquitous use of continuous integration systems. Increasingly powerful IDEs provide code searching and indexing capabilities, as well as small scale refactoring tools and integration with other SE tools. An understanding of the nature of these tools is broadly valuable - especially version control systems.

[1 CS Core hour, 3 KA Core hours]

Topics:

[CS Core]

- Software configuration management and version control
    - Configuration in version control, reproducible builds/configuration
    - Version control branching strategies. Development branches vs. release branches. Trunk-based development.
    - Merging/rebasing strategies, when relevant.

[KA Core]

- Release management
- Testing tools including static and dynamic analysis tools
- Software process automation
    - Build systems - the value of fast, hermetic, reproducible builds, compare/contrast approaches to building a project
    - Continuous Integration (CI) - the use of automation and automated tests to do preliminary validation that the current head/trunk revision builds and passes (basic) tests
    - Dependency management - updating external/upstream dependencies, package management, SemVer
- Design and communication tools (docs, diagrams, common forms of design diagrams like UML)
- Tool integration concepts and mechanisms
- Use of modern IDE facilities - debugging, refactoring, searching/indexing, etc.

Learning Outcomes:

[CS Core]

1. Describe the difference between centralized and distributed software configuration management.
2. Describe how version control can be used to help manage software release management.
3. Identify configuration items and use a source code control tool in a small team-based project.

[KA Core]

1. Describe how available static and dynamic test tools can be integrated into the software development environment.
2. Understand the use of CI systems as a ground-truth for the state of the team's shared code (build and test success).
3. Describe the issues that are important in selecting a set of tools for the development of a particular software system, including tools for requirements tracking, design modeling, implementation, build automation, and testing.
4. Demonstrate the capability to use software tools in support of the development of a software product of medium size.

SE/Product Requirements

Knowing how to build something is of little help if we do not know what to build. Product Requirements (aka Requirements Engineering, Product Design, Product Requirements solicitation, PRDs, etc.) introduces students to the processes surrounding the specification of the broad requirements governing development of a new product or feature.

[2 KA Core hours]

Topics:

[KA Core]

- Describe functional requirements using, for example, use cases or user stories

    - Using at least one method of documenting and structuring functional requirements
    - Understanding how the method supports design and implementation
    - Strengths and weaknesses of using a particular approach
- Properties of requirements including consistency, validity, completeness, and feasibility
- Requirements elicitation
    - Sources of requirements, for example, users, administrators, or support personnel
    - Methods of requirement gathering, for example, surveys, interviews, or behavioral analysis
- Non-functional requirements, for example, security, usability, or performance (aka Quality Attributes)
    - Cross reference IAS/Secure Software Engineering
- Risk identification and management
- Communicating and/or formalizing requirement specifications

[Non-core]

- Prototyping
    - A tool for both eliciting and validating/confirming requirements
- Product evolution
    - When requirements change, how to understand what effect that has and what changes need to be made
- Effort estimation
    - Learning techniques for better estimating the effort required to complete a task
    - Practicing estimation and comparing to how long tasks actually take
    - Effort estimation is quite difficult, so students are likely to be way off in many cases, but seeing the process play out with their own work is valuable

Illustrative Learning Outcomes:

[KA Core]

1. Compare different methods of eliciting requirements along multiple axes.
2. Identify differences between two methods of describing functional requirements (e.g. customer interviews, user studies, etc.) and the situations where each would be preferred.
3. Identify which behaviors are required, allowed, or barred from a given set of requirements and a list of candidate behaviors.
4. Collect a set of requirements for a simple software system.
5. Identify areas of a software system that need to be changed, given a description of the system and a set of new requirements to be implemented.
6. Identify the functional and non-functional requirements in a set of requirements.

[Non-core]

1. Create a prototype of a software system to validate a set of requirements. (Building a mock-up, MVP, etc.)
2. Estimate the time to complete a set of tasks, then compare estimates to the actual time taken.
3. Determine an implementation sequence for a set of tasks, adhering to dependencies between them, with a goal to retire risk as early as possible.
4. Write a requirement specification for a simple software system.

SE/Software Design

While Product Requirements focuses on the user-facing functionality of a software system, Software Design focuses on the engineer-facing design of internal software components. This encompasses large design concerns such as software architecture, as well as small-scale design choices like API design.

[1 CS Core hours; 4 KA Core hours]

Topics:

[CS Core]

- System design principles
    - Levels of abstraction (e.g. architectural design and detailed design)
    - Separation of concerns
    - Information hiding
    - Coupling and cohesion
- Software architecture
    - Design paradigms
        - Top-down functional decomposition / layered design
        - Data-oriented architecture
        - Object-oriented analysis and design
        - Event-driven design
    - Standard architectures (e.g. client-server, n-layer, pipes-and-filters, Model View Controller)
    - Identifying component boundaries and dependencies
- Programming in the large vs. programming in the small
- Code smells and other indications of code quality, distinct from correctness.

[KA Core]

- API design principles
    - Consistency
        - Consistent APIs are easier to learn and less error-prone
        - Consistency is both internal (between different portions of the API) and external (following common API patterns)
    - Composability
    - Documenting contracts
        - API operations should describe their effect on the system, but not generally their implementation
        - Preconditions, postconditions, and invariants
    - Expandability
        - Cross reference SE/Refactoring and Code Evolution
    - Error reporting
        - Errors should be clear, predictable, and actionable
        - Input that does not match the contract should produce an error
        - Errors that can be reliably managed without reporting should be managed
- Identifying and codifying data invariants and time invariants
- Structural and behavioral models of software designs
- Data design
    - Data structures

    - Storage systems
    - Cross reference Information Management, particularly IM/Data Modeling
- Requirement traceability
    - Understanding which requirements are satisfied by a design

. [Non-core]

- Design modeling, for instance with class diagrams, entity relationship diagrams, or sequence diagrams
- Measurement and analysis of design quality
- Principles of secure design and coding (cross reference IAS/Principles of Secure Design)
    - Principle of least privilege
    - Principle of fail-safe defaults
    - Principle of psychological acceptability
- Evaluating design tradeoffs (e.g. efficiency vs. reliability, security vs. usability)

Illustrative Learning Outcomes:

[CS Core]

1. Identify the standard software architecture of a given high-level design.
2. Select and use an appropriate design paradigm to design a simple software system and explain how system design principles have been applied in this design.
3. Adapt a flawed system design to better follow principles such as separation of concerns or information hiding.
4. Identify the dependencies among a set of software components in an architectural design.

[KA Core]

1. Design an API for a single component of a large software system, including identifying and documenting each operation's invariants, contract, and error conditions.
2. Evaluate an API description in terms of consistency, composability, and expandability.
3. Expand an existing design to include a new piece of functionality.
4. Design a set of data structures to implement a provided API surface.
5. Identify which requirements are satisfied by a provided software design.

[Non-core]

1. Translate a natural language software design into class diagrams.
2. Adapt a flawed system design to better follow the principles of least privilege and fail-safe defaults.
3. Contrast two software designs across different qualities, such as efficiency or usability.

SE/Software Construction

Software Construction focuses on practices that influence the direct production of software: use of tests, test driven development, coding style. More advanced topics extend into secure coding, dependency injection, work prioritization, etc.

[1 CS Core hour, 3 KA Core hours]

Topics:

[CS Core]

- Practical small-scale testing
    - Unit testing
    - Test-driven development - This is particularly valuable for students psychologically, as it is far easier to engage constructively with the challenge of identifying challenging inputs for a given API (edge cases, corner cases) a priori. If they implement first, the instinct is often to avoid trying to crash their new creation, while a test-first approach gives them the intellectual satisfaction of spotting the problem cases and then watching as more tests pass during the development process.
- Documentation
    - Interface documentation - Describe interface requirements, potentially including (formal or informal) contracts, pre and post conditions, invariants.
    - Implementation documentation should focus on tricky and non-obvious pieces of code, whether because the code is using advanced language features or the behavior of the code is complex. (Do not add comments that re-state common/obvious operations and simple language features.)
        - Clarify dataflow, computation, etc., focusing on what the code is
        - Identify subtle/tricky pieces of code and refactor to be self-explanatory if possible, or provide appropriate comments to clarify.

[KA Core]

- Coding style
    - Style guides
    - Commenting
    - Naming
- “Best Practices” for coding: techniques, idioms/patterns, mechanisms for building quality programs (cross reference IAS/Defensive Programming; SDF/Development Methods)
    - Defensive coding practices
    - Secure coding practices and principles
    - Using exception handling mechanisms to make programs more robust, fault-tolerant
- Debugging
- Use of libraries and frameworks developed by others

[Non-core]

- Larger-scale testing
    - Test doubles (stubs, mocks, fakes)
    - Dependency injection
- Work sequencing, including dependency identification, milestones, and risk retirement
    - Dependency identification: Identifying the dependencies between different tasks
    - Milestones: A collection of tasks that serve as a marker of progress when completed. Ideally, the milestone encompasses a useful unit of functionality.
    - Risk retirement: Identifying what elements of a project are risky and prioritizing completing tasks that address those risks
- Potential security problems in programs
    - Buffer and other types of overflows
    - Race conditions

    - Improper initialization, including choice of privileges
    - Input validation
- Documentation (autogenerated)
- Development context: “green field” vs. existing code base
    - Change impact analysis
    - Change actualization
- Release management

Learning Outcomes:

[CS Core]

- Write appropriate unit tests for a small component (several functions, a single type, etc.).
- Write appropriate interface and (if needed) implementation comments for a small component.

[KA Core]

- Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness.
- Write robust code using exception handling mechanisms.
- Describe secure coding and defensive coding practices.
- Select and use a defined coding standard in a small software project.
- Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration.
- Describe the process of analyzing and implementing changes to code base developed for a specific project.
- Describe the process of analyzing and implementing changes to a large existing code base.

. [Non-core]

- Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions.
- Write a software component that performs some non-trivial task and is resilient to input and run-time errors.

SE/Software Verification and Validation

Software Verification and Validation focuses on how to improve the value of testing - understand the role of testing, failure modes, and differences between good tests and poor ones.

[1 CS Core hour; 3 KA Core hours]

Topics:

[CS Core]

- Verification and validation concepts
    - Verification: Are we building the thing right?
    - Validation: Did we build the right thing?

- Why testing matters
    - Does the component remain functional as the code evolves?
- Testing objectives
    - Usability
    - Reliability
    - Conformance to specification
    - Performance
    - Security (cross reference IAS/Secure Software Engineering)
- Test kinds
    - Unit
    - Integration
    - Validation
    - System
- Stylistic differences between tests and production code
    - DAMP vs. DRY - more duplication is warranted in test code.

[KA Core]

- Test planning and generation
    - Test case generation, from formal models, specifications, etc.
    - Test coverage
        - Test matrices
        - Code coverage (how much of the code is tested)
        - Environment coverage (how many hardware architectures, OSes, browsers, etc. are tested)
    - Test data and inputs
- Test development (cross reference SDF/Development Methods)
    - Test-driven development (cross reference SE/Software Construction)
    - Object oriented testing, mocking, and dependency injection
    - Black-box and white-box testing techniques
    - Test tooling, including code coverage, static analysis, and fuzzing
- Verification and validation in the development cycle
    - Code reviews (cross reference SE/Software Construction)
    - Test automation, including automation of tooling
    - Pre-commit and post-commit testing
    - Trade-offs between test coverage and throughput/latency of testing
    - Defect tracking and prioritization
        - Reproducibility of reported defects
- Domain specific verification and validation challenges
    - Performance testing and benchmarking
    - Asynchrony, parallelism, and concurrency
    - Safety-critical
    - Numeric

[Non-core]

- Verification and validation tooling and automation
    - Static analysis
    - Code coverage
    - Fuzzing
    - Dynamic analysis and fault containment (sanitizers, etc.)
    - Fault logging and fault tracking
- Test planning and generation

    - Fault estimation and testing termination including defect seeding
    - Use of random and pseudo random numbers in testing
- Performance testing and benchmarking
    - Throughput and latency
    - Degradation under load (stress testing, FIFO vs. LIFO handling of requests)
    - Speedup and scaling
        - Amadhl's law
        - Gustafson's law
        - Soft and weak scaling
    - Identifying and measuring figures of merits
    - Common performance bottlenecks
        - Compute-bound
        - Memory-bandwidth bound
        - Latency-bound
    - Statistical methods and best practices for benchmarking
        - Estimation of uncertainty
        - Confidence intervals
    - Analysis and presentation (graphs, etc.)
    - Timing techniques
- Testing asynchronous, parallel, and concurrent systems
- Verification and validation of non-code artifacts (documentation, training materials)

Learning Outcomes:

[CS Core]

1. Explain why testing is important.
2. Distinguish between program validation and verification.
3. Describe different objectives of testing.
4. Compare and contrast the different types and levels of testing (regression, unit, integration, systems, and acceptance).

[KA Core]

1. Describe techniques for creating a test plan and generating test cases.
2. Create a test plan for a medium-size code segment which includes a test matrix and generation of test data and inputs.
3. Implement a test plan for a medium-size code segment.
4. Identify the fundamental principles of test-driven development methods and explain the role of automated testing in these methods.
5. Discuss issues involving the testing of object-oriented software.
6. Describe mocking and dependency injection and their application.
7. Undertake, as part of a team activity, a code review of a medium-size code segment.
8. Describe the role that tools can play in the validation of software.
9. Automate testing in a small software project.
10. Explain the roles, pros, and cons of pre-commit and post-commit testing.
11. Discuss the tradeoffs between test coverage and test throughput/latency and how this can impact verification.
12. Use a defect tracking tool to manage software defects in a small software project.
13. Discuss the limitations of testing in certain domains.

[Non-core]

1. Describe and compare different tools for verification and validation.
2. Automate the use of different tools in a small software project.
3. Explain how and when random numbers should be used in testing.
4. Describe approaches for fault estimation.
5. Estimate the number of faults in a small software application based on fault density and fault seeding.
6. Describe throughput and latency and provide examples of each.
7. Explain speedup and the different forms of scaling and how they are computed.
8. Describe common performance bottlenecks.
9. Describe statistical methods and best practices for benchmarking software.
10. Explain techniques for and challenges with measuring time when constructing a benchmark.
11. Identify the figures of merit, construct and run a benchmark, and statistically analyze and visualize the results for a small software project.
12. Describe techniques and issues with testing asynchronous, concurrent, and parallel software.
13. Create a test plan for a medium-size code segment which contains asynchronous, concurrent, and/or parallel code, including a test matrix and generation of test data and inputs.
14. Describe techniques for the verification and validation of non-code artifacts.

SE/Refactoring and Code Evolution

[2 KA Core hour]

Topics:

[KA Core]

- Hyrum's Law / The Law of Implicit Interfaces
- Backward compatibility
    - Compatibility is not a property of a single entity, it's a property of a *relationship*.
    - Backward compatibility needs to be evaluated in terms of provider + consumer(s) or with a well-specified model of what forms of compatibility a provider aspires to / promises.
- Refactoring
    - Standard refactoring patterns (rename, inline, outline, etc.)
    - Use of refactoring tools in IDE
    - Application of static-analysis tools (to identify code in need of refactoring, generate changes, etc.)
- Versioning
    - Semantic Versioning (SemVer)
    - Trunk-based development

[Non-core]

- "Large Scale" Refactoring - techniques when a refactoring change is too large to commit safely (large projects), or when it is impossible to synchronize change between provider + all consumers (multiple repositories, consumers with private code).
    - Express both old and new APIs so that they can co-exist
    - Minimize the size of *behavior* changes
    - Why these techniques are required, (e.g. "API consumers I can see" vs "consumers I can't see")

Illustrative Learning Outcomes:

1. Identify both explicit and implicit behavior of an interface, and identify potential risks from Hyrum's Law
2. Consider inputs from static analysis tools and/or Software Design principles to identify code in need of refactoring.
3. Identify changes that can be broadly considered "backward compatible," potentially with explicit statements about what usage is or is not supported
4. Refactor the implementation of an interface to improve design, clarity, etc. with minimal/zero impact on existing users
5. Evaluate whether a proposed change is sufficiently safe given the versioning methodology in use for a given project
6. [Non-core] Plan a complex multi-step refactoring to change default behavior of an API safely.

SE/Software Reliability

[2 KA Core hours]

Topics:

[KA Core]

- Concept of reliability as probability of failure or mean time between failures, and faults as cause of failures
- Identifying reliability requirements for different kinds of software (cross-reference SEP)
- Software failures caused by defects/bugs, and so for high reliability goal is to have minimum defects - by injecting fewer defects (better training, education, planning), and by removing most of the injected defects (testing, code review, etc.)
- Software reliability, system reliability and failure behavior (cross-reference SF/Reliability Through Redundancy)
- Defect injection and removal cycle, and different approaches for defect removal
- Compare the "error budget" approach to reliability with the "error-free" approach, and identify domains where each is relevant

[Non-core]

- Software reliability models
- Software fault tolerance techniques and models
    - Contextual differences in fault tolerance (e.g. crashing a flight critical system is strongly avoided, crashing a data processing system before corrupt data is written to storage is highly valuable)
- Software reliability engineering practices - including reviews, testing, practical model checking
- Identification of dependent and independent failure domains, and their impact on system reliability
- Measurement-based analysis of software reliability - telemetry, monitoring and alerting, dashboards, release qualification metrics, etc.

Learning Outcomes:

[KA Core]

1. Describe how to determine the level of reliability required by a software system.
2. Explain the problems that exist in achieving very high levels of reliability.
3. Understand approaches to minimizing faults that can be applied at each stage of the software lifecycle.

[Non-core]

4. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system.
5. Identify methods that will lead to the realization of a software architecture that achieves a specified level of reliability.
6. Identify ways to apply redundancy to achieve fault tolerance.
7. Identify single-point-of-failure (SPF) dependencies in a system design.

SE/Formal Methods

[Non-core]

The topics listed below have a strong dependency on core material from the Discrete Structures area, particularly knowledge units DS/Basic Logic and DS/Proof Techniques.

***Topics*:**

- Formal specification of interfaces
  - Specification of pre- and post- conditions
  - Formal languages for writing and analyzing pre- and post-conditions.
- Problem areas well served by formal methods
  - Lock-free programming, data races
  - Asynchronous and distributed systems, deadlock, livelock, etc.
- Comparison to other tools and techniques for defect detection
  - Testing
  - Fuzzing
- Formal approaches to software modeling and analysis
  - Model checkers
  - Model finders

Illustrative Learning Outcomes:

1. Describe the role formal specification and analysis techniques can play in the development of complex software and compare their use as validation and verification techniques with testing.
2. Apply formal specification and analysis techniques to software designs and programs with low complexity.
3. Explain the potential benefits and drawbacks of using formal specification languages.

Professional Dispositions

- **Collaborative**: Software engineering is increasingly described as a “team sport” - successful software engineers are able to work with others effectively. Humility, respect, and trust underpin the collaborative relationships that are essential to success in this field.

- **Professional**: Software engineering produces technology that has the chance to influence literally billions of people. Awareness of our role in society, strong ethical behavior, and commitment to respectful day-to-day behavior outside of one's team are essential.
- **Communicative:** No single software engineer on a project is likely to know all of the project details. Successful software projects depend on engineers communicating clearly and regularly in order to coordinate effectively.
- **Meticulous:** Software engineering requires attention to detail and consistent behavior from everyone on the team. Success in this field is clearly influenced by a meticulous approach - comprehensive understanding, proper procedures, and a solid avoidance of cutting corners.
- **Accountable:** The collaborative aspects of software engineering also highlight the value of accountability. Failing to take responsibility, failing to follow through, and failing to keep others informed are all classic causes of team friction and bad project outcomes.

Math Requirements

Desirable:

- Introductory statistics (performance comparisons, evaluating experiments, interpreting survey results, etc.)

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Professionalism / teamwork with Society, Ethics, and Professionalism (TODO)
- Data Modeling with Information Management
- Secure Programming with Information Security

Crosscutting themes:

- Ethics
- Programming

Course Packaging Suggestions

Advanced Course to include at least the following:

- Teamwork - 4 hours
- Tools and Environments - 4 hours
- Product Requirements - 2 hours
- Software Design - 5 hours
- Software Construction - 4 hours
- Refactoring and Code Evolution - 2 hours

- Software Reliability - 2 hours
- Professional Ethics and Professional Communication from SEP Knowledge Area: 7 hours

Pre-requisites:

- SDF - Sufficient programming fluency to focus on collaboration and process

Skill statement: A student who completes this course should be able to perform good quality code review for colleagues (especially focusing on professional communication and teamwork needs), read and write unit tests, use basic software tools (IDEs, version control, static analysis tools) and perform basic activities expected of a new hire on a software team.

Competency Specifications

- **Task 1:** Automate testing of new code under development
- **Competency Statement:** Break down the intended behavior of an API into one or more relevant inputs that can be used to ensure that behavior remains functional over time.
- **Competency area:** Software / Application
- **Competency unit:** Requirements / Design / Development / Testing / Documentation
- **Required knowledge areas and knowledge units:**
  - SDF / Development Methods
  - SE / Software Construction
- **Required skill level:** Apply, Develop
- **Core level:**

- **Task 2:** Perform a code review for a teammate
- **Competency Statement:** Communicate clearly and collaboratively to provide feedback to a teammate about a piece of code.
- **Competency area:** Software / Application
- **Competency unit:** Design / Documentation / Improvement
- **Required knowledge areas and knowledge units:**
  - SDF / Development Methods
  - SE / Teamwork
  - SE / Software Verification and Validation
- **Required skill level:** Apply
- **Core level:**

- **Task 3:** Document an API
- **Competency Statement:** Identify relevant APIs and create appropriate documentation to clarify correct usage and expected behavior.
- **Competency area:** Software

- **Competency unit:** Requirements / Design / Development / Documentation
- **Required knowledge areas and knowledge units:**
    - SDF / Fundamental Programming Concepts
    - SE / Software Construction
    - SE / Software Design
- **Required skill level:** Apply
- **Core level:**

- **Task 4:** Debug and/or modify some code written by others
- **Competency Statement:** Code will be read more often than it is written, and thus students should be well-versed in reading code, especially with appropriate scrutiny while debugging.
- **Competency area:** Software / Application
- **Competency unit:** Development / Evaluation / Maintenance / Improvement
- **Required knowledge areas and knowledge units:**
    - SDF / Development Methods
    - SE / Software Construction
- **Required skill level:** Apply
- **Core level:**

- **Task 5:** Identify appropriate tools to assist in development, design, or debugging
- **Competency Statement:** Be aware of common classes of software tools (static analysis, dynamic analysis, version control, coverage, refactoring, etc) and be able to identify problems where application of such tools would be appropriate.
- **Competency area:** Software / Application
- **Competency unit:** Development / Integration
- **Required knowledge areas and knowledge units:**
    - SDF / Development Methods
    - SE / Tools and Environments
    - SE / Software Construction
    - SE / Software Verification and Validation
- **Required skill level:** Explain
- **Core level:**

- **Task 6:** Refactor an API
- **Competency Statement:** Be capable of refactoring an API in use by others, identifying risky changes to syntax or semantics, and communicating a plan for how to make said changes safely (including use of relevant tools or versioning strategies)
- **Competency area:** Software / Application
- **Competency unit:** Design / Maintenance
- **Required knowledge areas and knowledge units:**

    - SE / Refactoring and Code Evolution
- **Required skill level:** Apply
- **Core level:**

- **Task 7:** Consistently format source code
- **Competency Statement:** Be familiar with common style guides and auto-formatting tools, as well as standard ideas around the value of consistency.
- **Competency area:** Software / Application
- **Competency unit:** Development / Maintenance
- **Required knowledge areas and knowledge units:**
    - SDF / Fundamental Programming Concepts
    - SE / Software Construction
- **Required skill level:** Apply
- **Core level:**

- **Task 8:** Design an API
- **Competency Statement:** Be able to design one or more related APIs, given requirements, with appropriate consideration for tradeoffs and necessary/desired quality attributes
- **Competency area:** Software
- **Competency unit:** Requirements / Design
- **Required knowledge areas and knowledge units:**
    - SE / Software Design
- **Required skill level:** Develop
- **Core level:**

- **Task 9:** Document a design decision
- **Competency Statement:** Understand that design decisions involve tradeoffs among various quality attributes (performance, usability, maintainability, readability, etc), and should be capable of explaining the tradeoffs and rationale behind important design choices in a software system
- **Competency area:** Software / Application
- **Competency unit:** Requirements / Design / Documentation
- **Required knowledge areas and knowledge units:**
    - SE / Software Design
- **Required skill level:** Develop
- **Core level:**

- **Task 10:** Work on a team effectively

- **Competency Statement:** Be capable of being a productive member of a team or other working group. Communication skills and a focus on long-term team dynamics are essential.
- **Competency area:** Application
- **Competency unit:** Documentation / Adaptation to social issues / Improvement
- **Required knowledge areas and knowledge units:**
  - SE / Teamwork
- **Required skill level:** Apply
- **Core level:**

- **Task 11:** Participate in design and development of an open-ended project
- **Competency Statement:** Work with others to achieve project goals specified by (potentially) non-technical clients, accounting for schedule and resource constraints.
- **Competency area:** Software / Application
- **Competency unit:** Requirements / Design / Documentation
- **Required knowledge areas and knowledge units:**
  - SE / Teamwork
  - SE / Software Design
  - SE / Software Construction
- **Required skill level:** Apply
- **Core level:**

Committee

Chair: Titus Winters (Google, New York City, NY, USA)

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Adam Vartanian, Cord, London, UK
- Bryce Adelstein Lelbach, NVIDIA, New York City, NY, USA
- Patrick Servello, CIWRO, Norman, OK, USA
- Pankaj Jalote, IIIT-Delhi, Delhi, India
- Christian Servin, El Paso Community College, El Paso, TX, USA

Contributors:

- Hyrum Wright, Google, Pittsburgh, PA, USA
- Olivier Giroux, Apple, Cupertino, CA, USA
- Gennadiy Civil, Google, New York City, NY, USA

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
SE	Teamwork	● Effective communication ● Common causes of team conflict, and approaches for conflict resolution ● Cooperative programming ● Roles and responsibilities in a software team ● Team processes ● Importance of team diversity	Evaluate	CS	2
SE	Teamwork	● Interfacing with stakeholders, as a team ● Risks associated with physical, distributed, hybrid and virtual teams	Explain	KA	2
SE	Tools and Environments	● Software configuration management and version control	Evaluate	CS	1
SE	Tools and Environments	● Release management ● Testing tools including static and dynamic analysis tools ● Software process automation ● Design and communication tools (docs, diagrams, common forms of design diagrams like UML) ● Tool integration concepts and mechanisms ● Use of modern IDE facilities - debugging, refactoring, searching/indexing, etc.	Explain	KA	3
SE	Product Requirements	● Describe functional requirements using, for example, use cases or user stories ● Properties of requirements including consistency, validity, completeness, and feasibility ● Requirements elicitation ● Non-functional requirements, for example, security, usability, or performance (aka Quality Attributes) ● Risk identification and management ● Communicating and/or formalizing requirement specifications	Apply	KA	2

SE	Software Design	● System design principles ● Software architecture ● Programming in the large vs. programming in the small ● Code smells and other indications of code quality, distinct from correctness.	Explain	CS	1
SE	Software Design	● API design principles ● Identifying and codifying data invariants and time invariants ● Structural and behavioral models of software designs ● Data design ● Requirement traceability	Apply	KA	4
SE	Software Construction	● Practical small-scale testing ● Documentation	Apply	CS	1
SE	Software Design	● Coding Style ● “Best Practices” for coding ● Debugging ● Use of libraries and frameworks developed by others	Apply	KA	3
SE	Software Verification and Validation	● Verification and validation concepts ● Why testing matters ● Testing objectives ● Test kinds ● Stylistic differences between tests and production code	Explain	CS	1
SE SDF	Software Verification and Validation	● Test planning and generation ● Test development (see SDF) ● Verification and validation in the development cycle ● Domain specific verification and validation challenges	Explain	KA	4
SE	Refactoring and Code Evolution	● Hyrum’s Law ● Backward Compatibility ● Refactoring ● Versioning	Explain	KA	1
SE SEPSF	Software Reliabilit	● Concept of reliability ● Identifying reliability requirements (see SEP)	Explain	KA	4

	y	● Software failures vs. defect injection/detection ● Software reliability, system reliability and failure behavior (cross-reference SF/Reliability Through Redundancy) ● Defect injection and removal cycle , and different approaches for defect removal ● Compare the “error budget” approach to reliability with the “error-free” approach, and identify domains where each is relevant			

Specialized Platform Development (SPD)

Preamble

The Specialized Platform Development (SPD) Knowledge Area (KA) refers to at- tributes involving creating a software platform to address specific needs and requirements for particular areas. Specialized platforms, such as healthcare providers, financial institutions, or transportation companies, are tailored to meet specific needs. Developing a specialized platform typically involves several key stages, i.e., requirements, design, development, deployment, and maintenance.

Changes since CS 2013: *The SPD Beta Versions considered the following factors:*

- ***Emerging Computing Areas*** such as data science/analytics – that use multi- platforms to retrieve sensing data. Cybersecurity – involves protecting certain data extraction, recognizing protocols to protect network transfer ability, and manipulating it. Artificial intelligence and machine learning – use artifacts that retrieve information for robotics and drones to perform specific tasks. This continuous emergence of computing technology areas has increased the appetite to develop platforms communicating software with specialized environments. This need has also increased the need to develop specialized programming languages for these platforms, such as web and mobile development. The Interactive Computing Platform addresses the advent of Large Language Models (LLMs), such as OpenAI's ChatGPT, OpenAI's Codex, and GitHub's Copilot, in addition to other platforms that perform data analysis, and visualizations.
- **Industry needs and competencies** have created a high demand for developers on specialized platforms, such as mobile, web, robotics, embedded, and interactive. Some of the unique professional competencies obtained by current job descriptions relevant to this KA are:
    - *Create a mobile app that provides a consistent user experience across various devices, screen sizes, and operating systems.*
    - *Analyze people's experience using a novel peripheral for an immersive system facilitated using a head-mounted display and mixed reality, with attention to usability and accessibility specifications.*
    - *Build and optimize a secure web page for evolving business needs using a variety of appropriate programming languages.*
    - *Develop application programming interfaces (APIs) to support mobile functionality and remain current with the terminology, concepts, and best practices for coding mobile apps.*

- *Availability of devices and artifacts, such as raspberry PIs, Arduinos, and mobile devices. The low cost of microcontrollers and devices, such as robots using ROS that can perform specialized actions, has*

The consideration of these factors resulted in the following significant changes from the CS2013 version:

- **Renamed of the Knowledge Area name**: From Platform-Based Development (PBD) to Specialized Platform Development due to the specific needs of the already mentioned tasks. This particular KA is often called *software platform development* since the specialized development takes part in the software stages for multi-platform development.
- **Increase of Computer Science Core Hours**: Based on the already mentioned needs, the SPD beta version has increased the number of computer science course hours from 0 to 9. The KA subsets the web and mobile knowledge units (often the most closely related units in CS Core) into foundations and specialized platforms core hours to provide flexibility and adaptability. This division allows programs at different institutions to offer different interests, concentrations, or degrees that focus on different application areas, where many of these concepts intersect the CS core. Therefore, the *common aspects, web* and mobile foundations have concepts in many CS programs' core. Finally, the rest of the knowledge units permit the curriculum to have an extended and flexible number of KA core hours.
- **Renamed old knowledge units and incorporated new ones**: in the spirit of capturing the future technology and societal responsibilities, the Robotics, Embedded Systems, and Society, Ethics, and Professionalism (SEP) knowledge units were introduced in this version. These KU work harmoniously with other KAs, consistent with the topics and concepts covered in specific KAs' knowledge units.

Specialized platform development provides a deep understanding of a particular user group's needs and requirements and considers other knowledge areas that help design and build a platform that meets other KA's needs. Considering other KAs' needs, SPD helps to streamline workflows, improve efficiency, and drive innovation across the recommended curriculum discussed in CS2023.

Core Hours

Knowledge Units	**CS Core**	**KA Core**
SPD/Web Foundations	2	
SPD/Mobile Foundations	1	
SPD/Common Aspects		3
SPD/Web Platforms		*

SPD/Mobile Platforms		*
SPD/Robot Platforms		*
SPD/Embedded Platforms		*
SPD/Game Platforms		*
SPD/Interactive Computing Platforms		*
Total	**3**	

Knowledge Units

SPD/Introduction → Common Aspects/Shared Concerns

This unit aims to develop core concepts related to specialized platform development. Students shall recognize the need to develop for various specialized platforms and their corresponding applications, the programming languages used for these applications, and how to effectively use such languages.

- Topics
    a. Overview of platforms (e.g., Web, Mobile, Game, Industrial)
        i. Input/Sensors/Control Devices/Haptic devices
        ii. Resource constraints
            - Computational
            - Data storage
            - Communication
            - Societal, Compliance, Security, Uptime availability, fault tolerance
        iii. Output/Actuators/Haptic devices
    b. Programming via platform-specific Application Programming Interface (API) vs traditional application construction
    c. Overview of Platform Languages (e.g., Kotlin, Swift, C#, C++, Java, JavaScript, HTML5)
    d. Programming under platform constraints (e.g., available development tools, development)
    e. Techniques for learning and mastering a platform-specific programming language.
- Illustrative Learning Outcomes
    a. List the constraints of mobile programming
    b. Describe the three-tier model of web programming

c. Describe how the state is maintained in web programming
d. List the characteristics of scripting languages

SPD/Web Platforms

This unit aims to develop concepts relating to web platforms. Concepts include programing language features, web platforms, frameworks, security and privacy considerations, architecture, and storage solutions.

- Topics
    a. Web programming languages (e.g., HTML5, JavaScript, PHP, CSS)
    b. Web platforms, frameworks, or meta-frameworks
    c. Software as a Service (SaaS)
    d. Web standards such as document object model, accessibility
    e. Security and Privacy considerations
    f. Analyzing requirements for web applications
    g. Computing services (e.g., Amazon AWS, Microsoft Azure)
        i. Cloud Hosting
        ii. Scalability (e.g., Autoscaling, Clusters)
        iii. How to estimate costs for these services (based on requirements)
    h. Data management
        i. Data residency (where the data is located and what paths can be taken to access it)
        ii. Data integrity: guaranteeing data is accessible and guaranteeing that data is deleted when required
    i. Architecture
        i. Monoliths vs. Microservices
        ii. Micro-frontends
        iii. Event-Driven vs. RESTful architectures: advantages and disadvantages
        iv. Serverless, cloud computing on demand
    j. Storage Solutions
        i. Relational Databases
        ii. NoSQL databases

- Illustrative Learning Outcomes
    a. Design and Implement a web application using microservice architecture design.
    b. Describe the web platform's constraints and opportunities, such as hosting, services, and scalability, that developers should consider.
    c. Compare and contrast web programming with general-purpose programming.
    d. Describe the differences between Software-as-a-Service and traditional software products.
    e. Discuss how web standards impact software development.

f. Review an existing web application against a current web standard.

SPD/Mobile Platforms

This unit aims to develop concepts relating to web platform technologies and considerations.

The mobile platform also offers local, on-device computing. Typical on-device security and machine-language-specific chips make possible applications with different impacts from other traditional platforms.

- Topics
    a. Development
        i. Mobile programming languages
        ii. Mobile programming environments
        iii. Native versus cross-platform development
        iv. Software architecture patterns used in mobile development
    b. Mobile platform constraints
        i. User interface design
        ii. Understanding differences in user experience between mobile and web-based applications
        iii. Security
        iv. Power/performance tradeoff
    c. Access
        i. Accessing data through APIs
        ii. Designing API endpoints for mobile apps: pitfalls and design considerations
        iii. Network and the Web interfaces
    d. Mobile computing affordances
        i. Location-aware applications
        ii. Sensor-driven computing (e.g., gyroscope, accelerometer, health data from a watch)
        iii. Telephony, Instant messaging
        iv. Augmented Reality.
    e. Specification and Testing
    f. Asynchronous computing
        i. How it differs from traditional synchronous programming
        ii. Handling success via callbacks
        iii. Handling errors asynchronously
        iv. Testing asynchronous code and typical problems in testing

- Illustrative Learning Outcomes

a. Implement a location-aware mobile application that uses data APIs.
b. Implement a sensor-driven mobile application that logs data on a server.
c. Implement a communication app that uses telephony and instant messaging.
d. Compare and contrast mobile programming with general-purpose programming.
e. Describe the pros and cons of native and cross-platform mobile app development.

SPD/Robot Platforms

The robot platforms knowledge unit considers topics related to the deployment of software on existing robot platforms and the application of these robots. Concepts include robotic platforms, specialized programming languages, tools for robotic development, and the interconnection between physical and simulated systems.

- Topics
    a. Types of robotic platforms and devices
    b. Sensors, embedded computation, and effectors (actuators)
    c. Robot-specific languages and libraries
    d. Robotic platform constraints and design considerations
    e. Interconnections with physical or simulated systems
    f. Robotics
        i. Robotic software architecture (e.g., using the Robot Operating System)
        ii. Forward kinematics
        iii. Inverse kinematics
        iv. Dynamics
        v. Navigation and robotic path planning
        vi. Manipulation and grasping
        vii. Safety considerations

- Illustrative Learning Outcomes
    a. Design and implement an application on a given robotic platform (e.g., using Lego Mindstorms, MATLAB, or the Robot Operating System connected to a simulator or physical robot)
    b. Assemble an Arduino-based robot kit and program it to navigate a maze
    c. Compare robot-specific languages and techniques with those used for general-purpose software development
    d. Explain the rationale behind the design of the robotic platform and its interconnections with physical or simulated systems,
    e. Given a high-level application, design a robot software architecture using ROS specifying all components and interconnections (ROS topics) to accomplish that application
    f. Discuss the constraints a given robotic platform imposes on developers

SPD/Embedded Platforms

This Knowledge unit considers embedded computing platforms and their applications. Embedded platforms cover knowledge ranging from sensor technology to ubiquitous computing applications.

Reference PDC and OS for topics related to concurrency, timing, scheduling, and timeouts.

- Topics
    a. Introduction to the Unique Characteristics of Embedded Systems
        i. real-time vs. soft real-time and non-real-time systems
        ii. Resource constraints (e.g., memory profiles, deadlines, etc.)
    b. Safety considerations and safety analysis
    c. Sensors and Actuators
    d. Embedded programming
    e. Real-time resource management
    f. Analysis and Verification
    g. Application Design
- Illustrative Learning Outcomes
    a. Design and implement a small embedded system for a given platform (e.g., a smart alarm clock or a drone)
    b. Describe unique characteristics of embedded systems versus other systems
    c. Interface with sensors/actuators
    d. Debug a problem with an existing embedded platform
    e. Identify different types of embedded architectures
    f. Evaluate which architecture is best for a given set of requirements

SPD/Game Platforms

The Game Platforms knowledge unit draws attention to concepts related to the engineering of performant real-time interactive software on constrained computing platforms. Material on requirements, design thinking, quality assurance, and compliance enhances problem-solving skills and creativity.

- Topics
    a. Historical and Contemporary Platforms for Games
        i. *Evolution of Game Platforms*: Brown Box to Metaverse and beyond; Improvement in Computing Architectures (CPU and GPU); Platform Convergence and Mobility

ii. *Typical Game Platforms*: Personal Computer; Home Console; Handheld Console; Arcade Machine; Interactive Television; Mobile Phone; Tablet; Integrated Head-Mounted Display; Immersive Installations and Simulators; Internet of Things enabled Devices; CAVE Systems; Web Browsers; Cloud-based Streaming Systems
iii. *Characteristics and Constraints of Different Game Platforms*: Features (local storage, internetworking, peripherals); Run-time performance (GPU/CPU frequency, number of cores); Chipsets (physics processing units, vector co-processors); Expansion Bandwidth (PCIe); Network throughput (Ethernet); Memory types and capacities (DDR/GDDR); Maximum stack depth; Power consumption; Thermal design; Endian; etc.
iv. *Typical Sensors, Controllers, and Actuators:* typical control system designs—peripherals (mouse, keypad, joystick), game controllers, wearables, interactive surfaces; electronics and bespoke hardware; computer vision, inside-out tracking, and outside-in tracking; IoT-enabled electronics and i/o; etc.

b. Social, Legal, and Ethical Considerations for Game Platforms
i. *Usability*: user requirements; affordances; ergonomic design; user research; heuristic evaluation methods for games
ii. *Accessibility:* equality and access; universal design; legislated requirements for game platforms; compliance evaluation.
iii. *Sustainability:* materials; power usage; supply-chain; recycling; planned obsolescence; etc.

c. Real-time Simulation and Rendering Systems
i. *CPU and GPU architectures*: Flynn's taxonomy; parallelization; instruction sets; common components—graphics compute array, graphics memory controller, video graphics array basic input/output system; bus interface; power management unit; video processing unit; display interface, etc.
ii. *Pipelines for physical simulations and graphical rendering:* tile-based, immediate-mode, etc.
iii. *Common Contexts for Algorithms, Data Structures, and Mathematical Functions*: game loops; spatial partitioning, viewport culling, and level of detail; collision detection and resolution; physical simulation; behavior for intelligent agents; procedural content generation; etc.
iv. *Media representations, i/o, and computation techniques for virtual worlds:* audio; music; sprites; models and textures; text; dialogue; multimedia (e.g., olfaction, tactile); etc.

d. Game Development Tools and Techniques
i. *Programming Languages*: C++; C#; Lua; Python; JavaScript; etc.
ii. *Shader Languages*: HLSL; GLSL; ShaderGraph; etc.
iii. *Graphics Libraries and APIs*: DirectX; SDL; OpenGL; Metal; Vulkan; WebGL
iv. *Common Development Tools and Environments*: IDEs; Debuggers; Profilers; Version Control Systems (including those handling binary assets); Development Kits and Production/Consumer Kits; Emulators;

Engines—Open Game Engine; Unreal; Unity; Godot; CryEngine; Phyre; Source 2; Phaser; Twine; etc.

v. *Techniques*: Ideation; Prototyping; Iterative Design and Implementation; Compiling Executable Builds; Development Operations and Quality Assurance—Play Testing and Technical Testing; Profiling; Optimization; Porting; Internationalization and Localization; Networking; etc.

e. Game Design
   i. *Vocabulary:* game definitions; mechanics-dynamics-aesthetics model; industry terminology; models of experience and emotion; etc.
   ii. *Design Thinking and User-Centered Experience Design:* methods of designing games; iteration, incrementing, and the double-diamond; phases of pre- and post-production; stakeholder and customer involvement; community management.
   iii. *Genres:* Adventure; walking simulator; first-person shooter; real-time strategy; multiplayer online battle arena (MOBA); role-playing game (rpg); etc.
   iv. *Audiences and Player Taxonomies*: people who play games; diversity and broadening participation; pleasures, player types, and preferences; Bartle, yee, etc.
   v. *Proliferation of digital game technologies to domains beyond entertainment*: Education and Training; Serious Games; Virtual Production; Esports; Gamification; Immersive Experience Design; Creative Industry Practice; Artistic Practice; Procedural Rhetoric.

- Illustrative Learning Outcomes
   a. Recall the characteristics of common general-purpose graphics processing architectures
   b. Identify the key stages of the immediate-mode rendering pipeline
   c. Describe the key constraints a given game platform will likely impose on developers
   d. Translate complex mathematical functions into performant source code
   e. Use an industry-standard graphics API to render a 3D model in a virtual scene
   f. Modify a shader to change a visual effect according to stated requirements
   g. Implement a game for a particular platform according to a specification
   h. Optimize a function for processing collision detection in a simulated environment
   i. Assess a game's run-time and memory performance using an industry-standard tool and development environment
   j. Compare the interfaces of different game platforms, highlighting their respective implications for human-computer interaction
   k. Recommend an appropriate set of development tools and techniques for implementing a game of a particular genre for a given platform
   l. Discuss the key challenges in making a digital game that is cross-platform compatible
   m. Suggest how game developers can enhance the accessibility of a game interface
   n. Create novel forms of gameplay using frontier game platforms

SPD/Interactive Computing Platforms

This knowledge unit concerns interactive computing platforms and the use of Large Language Models (LLM) to interact with computer users based on queries and other interactivity actions. Most of these topics span applications for Data Science, Quantum Computing, and various creative disciplines. Additionally, it concentrates on LLM to interact with.

- Topics
    a. Data Analysis Platforms
        i. Jupyter notebooks; Google Colab; R; SPSS; Observable, etc.
        ii. Cloud SQL/data analysis platforms (e.g., BigQuery)
            - Apache Spark
    b. Data Visualizations
        i. Interactive presentations backed by data
        ii. Design tools requiring low-latency feedback loops
            - rendering tools
            - graphic design tools
    c. Creative coding
        i. Creative interactive frameworks (can crossover with web, embedded/IoT/other low-fidelity hardware)
            - Live Music
            - Generative Art
            - Exhibition/demonstrative works
        ii. Machine-assisted interactivity (e.g., AI/ML pairing)
    d. Large Language Models (LLMs)
        i. Use of applications such as OpenAI's ChatGPT, OpenAI's Codex, and GitHub's Copilot
- Supporting math studies:
    a. Signal analysis / Fourier analysis / Signal processing (for music composition, audio/RF analysis)
    b. Statistics (for Data Analysis)
- Supporting humanities studies
    a. Visual art
    b. Journalism and other interactive storytelling. Exploratory, data-intensive applications intended to be consumed by a wide audience.
    c. Music theory, composition

- **Illustrative Learning Outcomes**
    a. Interactively analyze large datasets
    b. Create a backing track for musical performance (e.g., with live coding)
    c. Create compelling computational notebooks that construct a narrative for a given journalistic goal/story.

d. Implement interactive code that uses a dataset and generates exploratory graphics
e. Create a program that performs a task using LLM systems
f. Contrast a program developed by an AI platform and by a human
g. Implement a system that interacts with a human without using a screen
h. Contextualize the attributes of different data analysis styles, such as interactive vs. engineered pipeline
i. Write a program using a notebook computing platform (e.g., searching, sorting, or graph manipulation)

SPD/Society, Ethics, and Professionalism

This knowledge unit captures the society, ethics, and professionalism aspects from the specialized platform development viewpoint. Every stage from the software development perspective impacts the SEP knowledge unit.

Topics

- Augmented technology and societal impact
- Robotic design
- Graphical User Interfaces considerations for DEI
- Recognizing data privacy and implications

Professional Dispositions

- Learning to learn (new platforms, languages)
- Inventiveness (in designing software architecture within non-traditional constraints)
- Adaptability (to new constraints)
- Learning to debug and test code

Math Requirements

Desired:

- Calculus
- Linear Algebra
- Probability/Statistics (e.g., dynamic systems, visualization e.g., algorithmically generated Tuftian-style displays)
- Discrete Math/Structures (e.g., graphs for process control and path search)

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Artificial Intelligence
- Graphics
- Human-Computer Interaction
- Modeling
- Programming Languages
- Software Engineering
- Requirements Engineering

Competency Specifications

- **Task 1:** Determine whether to develop an app natively or using cross-platform tools.
- **Competency Statement:** Have technical and app design knowledge, understand performance and scalability issues, and evaluate different approaches and tools by carefully considering factors such as app requirements, target audience, time-to-market, and costs.
- **Competency area:** Application
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units:**
    - SE / Tools and Environments
    - SPD / Common Aspects
    - SPD / Mobile Platform
- **Required skill level:** Explain
- **Core level:** CS core

- **Task 2:** Create a mobile app that provides a consistent user experience across various devices, screen sizes, and operating systems.
- **Competency Statement:** Have the technical knowledge and design skills for mobile app development, optimize the app's usability and performance, and conduct extensive testing to ensure its functionality on various devices and platforms.

- **Competency area:** Application
- **Competency unit:** Design / Development / Testing
- **Required knowledge areas and knowledge units:**
    - HCI / Understanding the User
    - PL / Object-Oriented Programming
    - SDP / Development Methods
    - SE / Tools and Environments
    - SE / Software Design
    - SE / Software Construction
    - SE / Software Verification and Validation
    - SPD / Mobile Platform
    - SEP / TBD
- **Required skill level:** Develop
- **Core level:** CS core

- **Task 3:** Using an engine, translate prototype gameplay implemented by a designer using blueprints into high-performance and maintainable code with attention to maintaining cross-platform compatibility.
- **Competency Statement:** Demonstrate an ability to implement, profile, and optimize software for a game platform based on set requirements and an ability to verify the functional coherence, performance, and portability of the solution
- **Competency area:** Software, Application
- **Competency unit:** Development, Testing, Evaluation
- **Required knowledge areas and knowledge units:**
    - SE / Software Design
    - SE / Software Construction
    - SE / Software Verification and Validation
    - AL / Algorithmic Strategies
    - SF / System Performance
    - SF / Performance Evaluation
    - PL / Object-Oriented Programming

    - SPD / Common Aspects
    - SPD / Game Platforms
- **Required skill level:** Develop
- **Core level:**

- **Task 4:** Analyze people's experience using a novel peripheral for an immersive system facilitated using a head-mounted display and mixed reality, with attention to usability and accessibility specifications.
- **Competency Statement:** Demonstrate sufficient capacity to assess the characteristics of a game platform interface for quality, legislative requirements, and end-user acceptance
- **Competency area:** Systems
- **Competency unit:** Requirements, Testing, Evaluation, Consumer Acceptance, Adaptation to Social Issues
- **Required knowledge areas and knowledge units:**
    - HCI / Understanding the User
    - HCI / Accessibility and Inclusive Design
    - HCI / Evaluating the Design
    - GIT / Immersion
    - GIT / Physical Computing
    - SE / Software Verification and Validation
    - SEP / Equity, Diversity, and Inclusion
    - SPD / Game Platforms
- **Required skill level:** Apply
- **Core level:**

- **Task 5:** Build and optimize a secure web page for evolving business needs using a variety of *appropriate* programming languages.
- **Competency Statement:** Have security and application design knowledge, understand potential security hazards and room for optimization.
- **Competency area:** Application
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units:**
- **Required skill level:** Develop

    - AR / Performance and Energy Efficiency
    - CYB /
    - NC / Network Security
    - OS / Protection and Safety
    - SF / System Security
    - SE / Software Design
    - SE / Tools and Environments
    - SPD / Common Aspects
    - SPD / Mobile Platform
    - SEP / Privacy
- **Core level:** CS core

- **Task 6:** Develop application programming interfaces (APIs) to support mobile functionality and remain current with the terminology, concepts, and best practices for coding mobile apps.
- **Competency Statement:** Manifest proficiency as mobile app developer by designing, developing, and implementing mobile apps. Developer employs application programming interfaces for code reduction and development.
- **Competency area:** Application
- **Competency unit:** Development/Evaluation
- **Required knowledge areas and knowledge units:**
    - FPL / Hardware Interface
    - OS / File Systems API and Implementation
    - SE / Software Design
    - SE / Tools and Environments
    - SPD / Common Aspects
    - SPD / Mobile Platform
- **Required skill level:** Develop
- **Core level:** CS core

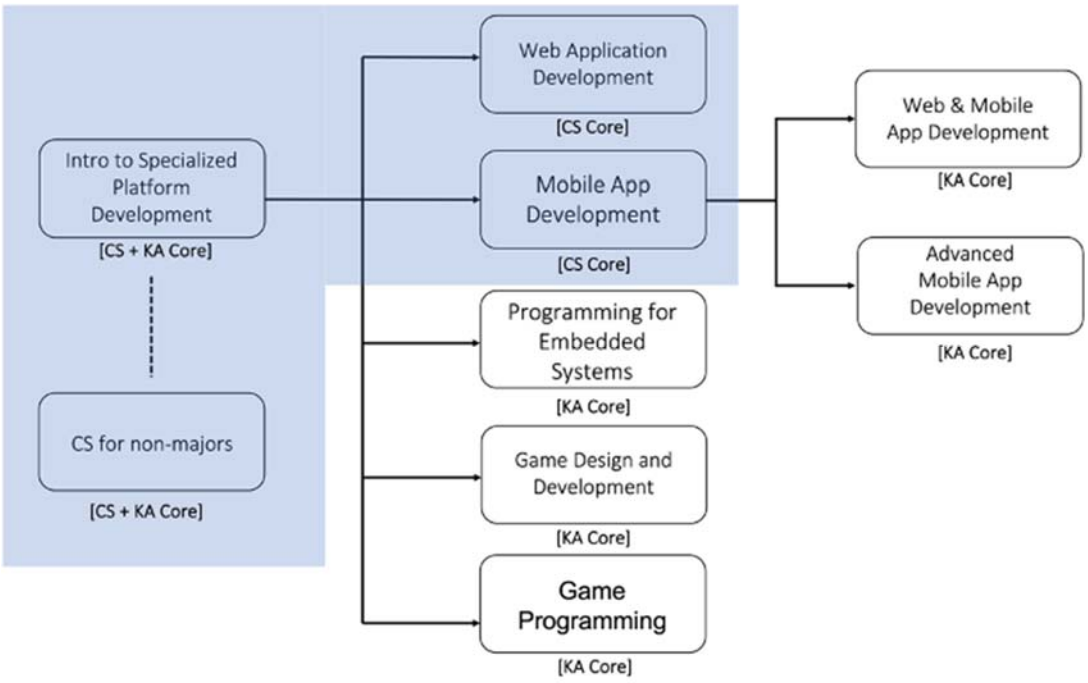
- **Task 7:** Identify robotic applications' or machines' purposes and goals and complete all design stages for systems that accomplish stated goals.****
- **Competency Statement:** Own understanding on interactivity between software and hardware interfacing to perform robotic-based applications.
- **Competency area:** Application
- **Competency unit:** Development/Evaluation
- **Required knowledge areas and knowledge units:**
    - AL
    - CYB/
    - SE / Tools and Environments
    - SE / Software Design
    - SPD / Common Aspects
    - SPD / Mobile Platform
- **Required skill level:** Explain
- **Core level:**

- **Task 8:** Design program architecture based on project requirements and hardware specifications by writing software code, embedded programs, and system protocols.
- **Competency Statement:** Indicate expertise recognizing embedded programming to various devices and interactively between libraries, platforms, and software.
- **Competency area:** Application
- **Competency unit:** Development/Deployment/Integration
- **Required knowledge areas and knowledge units:**
    - FPL/Hardware Interface
    - SE / Software Design
    - SE / Tools and Environments
    - SF / System Performance
    - SPD / Common Aspects
    - SPD / Embedded Systems
- **Required skill level:** Develop
- **Core level:**

- **Task 9:** Provide continued support for one or more web properties.
- **Competency Statement:** Indicate expertise recognizing embedded programming to various devices and interactively between libraries, platforms, and software.
- **Competency area:** Application
- **Competency unit:** Development/Deployment/Integration
- **Required knowledge areas and knowledge units:**
    - DM / NoSQL System
    - SE / Tools and Environments
    - SE / Software Design
    - SPD / Common Aspects
    - SPD / Web Development
    - NC / Single Hop Communication
    - OS / File Systems API and Implementation
- **Required skill level:** Apply
- **Core level:** CS core

- **Task 10:** Cooperating with back-end developers, designers, and the rest of the team to deliver well-architected and high-quality solutions.
- **Competency Statement:** Aptitude to discuss, synthesize, and integrate ideas from various departments related to product management.
- **Competency area:** Application
- **Competency unit:** Deployment/Integration
- **Required knowledge areas and knowledge units:**
    - SE / Project Management
    - SE / Software Design
    - SE / Teamwork
    - SPD / Common Aspects
    - SPD / Web Development
    - SPD / Mobile Development
- **Required skill level:** Explain
- **Core level:**

Course Packaging Suggestions



Committee

Chair: Christian Servin (El Paso Community College, El Paso, TX, USA)

Members:

- Sherif G. Aly, The American University in Cairo, Egypt
- Yoonsik Cheon, The University of Texas at El Paso, El Paso, Texas, USA
- Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA
- Claudia L. Guevara, Jochen Schweizer mydays Holding GmbH, Munich, Germany
- Larry Heimann, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
- Amruth N. Kumar, Ramapo College of New Jersey, Mahwah, NJ, USA
- R. Tyler Pirtle, Google
- Michael Scott, Falmouth University, UK

Contributors:

- Orlando Gordillo, NASA, USA
- Sean R. Piotrowski, Rider University, USA
- Mark 0'Neil, Blackboard Inc., USA

- John DiGennaro, Qwickly
- Rory K. Summerley, Falmouth University, Penryn, Cornwall, UK.

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
	Common Aspects	● Overview of platforms (e.g., Web, Mobile, Game, Robot, Embedded, and Interactive) ● Programming via platform-specific Application Programming Interface (API) vs traditional application construction ● Overview of Platform Languages (e.g., Kotlin, Swift, C#, C++, Java, JavaScript, HTML5) ● Programming under platform constraints (e.g., available development tools, development) ● Techniques for learning and mastering a platform-specific programming language.	Apply	CS	2
	Web Platforms	● Web programming languages (e.g., HTML5, JavaScript, PHP, CSS) ● Web platforms, frameworks, or meta-frameworks ● Software as a Service (SaaS) ● Web standards such as document object model, accessibility ● Security and Privacy considerations ● Analyzing requirements for web applications ● Computing services (e.g., Amazon AWS, Microsoft Azure) ● Data management ● Architecture ● Storage Solutions	Apply	CS	3
	Mobile Platforms	● Development ● Mobile platform constraints ● Access ● Mobile computing affordances ● Specification and Testing ● Asynchronous computing	Apply	CS	3
	Robot Platforms	● Types of robotic platforms and devices ● Sensors, embedded computation, and effectors (actuators) ● Robot-specific languages and libraries ● Robotic platform constraints and design	Apply	KA	3

		considerations ● Interconnections with physical or simulated systems ● Robotics			
	Embedded Platforms	● Introduction to the Unique Characteristics of Embedded Systems. ● Safety considerations and safety analysis ● Sensors and Actuators ● Embedded programming ● Real-time resource management ● Analysis and Verification ● Application Design	Apply	KA	3
	Game Platforms	● Historic and Contemporary Platforms for Games ● Social, Legal, and Ethical Considerations for Game Platforms ● Real-time Simulation and Rendering Systems ● Game Development Tools and Techniques ● Game Design	Apply	KA	4
	Interactive	● Data Analysis Platforms ● Data Visualizations ● Creative coding ● Quantum Computing Platforms ● Language Models (LLMs) ● Supporting math studies ● Supporting humanities studies	Apply	KA	2
	SEP	● TBD		KA	3

Systems Fundamentals (SF)

Preamble

A computer system is a set of hardware and software infrastructures upon which applications are constructed. Computer systems have become a pillar of people's daily life. As such, learning the knowledge about computer systems, grasping the skills to use and design these systems, and understanding the fundamental rationale and principles in computer systems are essential to equip students with the necessary competency toward a career related to computer science.

In the curriculum of computer science, the study of computer systems typically spans across multiple courses, including, but not limited to, operating systems, parallel and distributed systems, communications networks, computer architecture and organization and software engineering. The System Fundamentals knowledge area, as suggested by its name, focuses on the fundamental concepts in computer systems that are shared by these courses within their respective cores. The goal of this knowledge area is to present an integrative view of these fundamental concepts in a unified albeit simplified fashion, providing a common foundation for the different specialized mechanisms and policies appropriate to the particular domain area. These concepts include an overview of computer systems, basic concepts such as state and state transition, resource allocation and scheduling, and so on.

Changes since CS2013: Compared to CS2013, the SF knowledge area makes the following major changes to the knowledge units:

1. Added two new units: system security and system design;
2. Added a new unit of system performance, which includes the topics from the deprecated unit of proximity and the deprecated unit of virtualization and isolation;
3. Added a new unit of performance evaluation, which includes the topics from the deprecated unit of evaluation and the deprecated unit of quantitative evaluation;
4. Changed the unit of computational paradigms to overview of computer systems, deprecated some topics in the unit, and added topics from the deprecated unit of cross-layer communications;
5. Changed the unit of state and state transition to basic concepts, and added topics such as finite state machines;
6. Changed some topics in the unit of parallelism, such as simple application-level parallel processing;
7. Deprecated the unit of cross-layer communications, and moved parts of its topics to the unit of overview of computer systems;

8. Deprecated the units of evaluation and quantitative evaluation, and moved parts of their topics to the unit of performance evaluation;
9. Deprecated the units of proximity and virtualization and isolation, and moved parts of their topics to the unit of system performance;
10. Deprecated the units of parallelism, and moved parts of its topic to the unit of basic concepts;
11. Renamed the unit of reliability through redundancy to system reliability.

Core Hours

Knowledge Unit	CS Core	KA Core
Overview of Computer Systems	3	
Basic Concepts	4	
Resource Allocation and Scheduling	1	2
System Performance	2	2
Performance Evaluation	2	2
System Reliability	2	1
System Security	2	1
System Design	2	1
Total	**18**	**9**

Knowledge Units

SF/Overview of Computer Systems

[3 CS Core hours]

Topics:

- Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; datapath + control + memory)
- Hardware as a computational paradigm: Fundamental logic building blocks; Logic expressions, minimization, sum of product forms
- Programming abstractions, interfaces, use of libraries
- Distinction between application and OS services, remote procedure call

- Application-OS interaction
- Basic concept of pipelining, overlapped processing stages
- Basic concept of scaling: going faster vs. handling larger problems

Learning Outcomes:

1. Describe the basic building blocks of computers and their role in the historical development of computer architecture.
2. Design a simple logic circuit using the fundamental building blocks of logic design to solve a simple problem (e.g., adder).
3. Use tools for capture, synthesis, and simulation to evaluate a logic circuit design.
4. Describe how computing systems are constructed of layers upon layers, based on separation of concerns, with well-defined interfaces, hiding details of low layers from the higher layers.
5. Describe that hardware, OS, VM, application are additional layers of interpretation/processing.
6. Describe the mechanisms of how errors are detected, signaled back, and handled through the layers.
7. Construct a simple program (e.g., a TCP client/server) using methods of layering, error detection and recovery, and reflection of error status across layers.
8. Find bugs in a layered program by using tools for program tracing, single stepping, and debugging.
9. Understand the concept of strong vs. weak scaling, i.e., how performance is affected by scale of problem vs. scale of resources to solve the problem. This can be motivated by simple, real-world examples.

SF/Basic Concepts
[4 CS Core hours]
Topics:

- Digital vs. Analog/Discrete vs. Continuous Systems
- Simple logic gates, logical expressions, Boolean logic simplification
- Clocks, State, Sequencing
- State and state transition (e.g., starting state, final state, life cycle of states)
- Finite state machines (e.g., NFA, DFA)
- Combinational Logic, Sequential Logic, Registers, Memories
- Computers and Network Protocols as examples of State Machines
- Sequential vs. parallel processing
- Application-level sequential processing: single thread
- Simple application-level parallel processing: request level (web services/client-server/distributed), single thread per server, multiple threads with multiple servers, pipelining

Learning Outcomes:

1. Describe the differences between digital and analog systems, and between discrete and continuous systems. Can give real-world examples of these systems.
2. Describe computations as a system characterized by a known set of configurations with transitions from one unique configuration (state) to another (state).
3. Describe the distinction between systems whose output is only a function of their input (stateless) and those with memory/history (stateful).
4. Develop state machine descriptions for simple problem statement solutions (e.g., traffic light sequencing, pattern recognizers).
5. Describe a computer as a state machine that interprets machine instructions.
6. Explain how a program or network protocol can also be expressed as a state machine, and that alternative representations for the same computation can exist.
7. Derive time-series behavior of a state machine from its state machine representation (e.g., TCP connection management state machine).
8. Write a simple sequential problem and a simple parallel version of the same program.
9. Evaluate the performance of simple sequential and parallel versions of a program with different problem sizes, and be able to describe the speed-ups achieved.
10. Demonstrate on an execution timeline that parallelism events and operations can take place simultaneously (i.e., at the same time). Explain how work can be performed in less elapsed time if this can be exploited.

SF/Resource Allocation and Scheduling
[1 CS Core hour and 2 KA Core hours]
Topics:

- Different types of resources (e.g., processor share, memory, disk, net bandwidth)
- Common scheduling algorithms (e.g., first-come-first-serve scheduling, priority-based scheduling, fair scheduling and preemptive scheduling)
- Advantages and disadvantages of common scheduling algorithms

Learning Outcomes:

1. Define how finite computer resources (e.g., processor share, memory, storage and network bandwidth) are managed by their careful allocation to existing entities.
2. Describe how common scheduling algorithms work.
3. Describe the pros and cons of common scheduling algorithms
4. Implement common scheduling algorithms, and evaluate their performances.

SF/System Performance
[2 CS Core hours and 2 KA Core hours]
[Cross-reference: AR/Memory Management, OS/Virtual Memory]

Topics:

- Latencies in computer systems
    - Speed of light and computers (one foot per nanosecond vs. one GHz clocks)
    - Memory vs. disk latencies vs. across the network memory
- Caches and the effects of spatial and temporal locality on performance in processors and systems
- Caches and cache coherency in databases, operating systems, distributed systems, and computer architecture
- Introduction into the processor memory hierarchy and the formula for average memory access time
- Rationale of virtualization and isolation: protection and predictable performance
- Levels of indirection, illustrated by virtual memory for managing physical memory resources
- Methods for implementing virtual memory and virtual machines

***Learning Outcomes**:*

1. Explain the importance of locality in determining system performance.
2. Calculate average memory access time and describe the tradeoffs in memory hierarchy performance in terms of capacity, miss/hit rate, and access time.
3. Explain why it is important to isolate and protect the execution of individual programs and environments that share common underlying resources.
4. Describe how the concept of indirection can create the illusion of a dedicated machine and its resources even when physically shared among multiple programs and environments.
5. Measure the performance of two application instances running on separate virtual machines, and determine the effect of performance isolation.

SF/Performance Evaluation
[2 CS Core hours and 2 KA Core hours]
***Topics**:*

- Performance figures of merit
- Workloads and representative benchmarks, and methods of collecting and analyzing performance figures of merit
- CPI (Cycles per Instruction) equation as tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations.
- Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can
- Analytical tools to guide quantitative evaluation
- Order of magnitude analysis (Big O notation)
- Analysis of slow and fast paths of a system
- Events on their effect on performance (e.g., instruction stalls, cache misses, page faults)
- Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation
- Microbenchmarking pitfalls

***Learning Outcomes**:*

1. Explain how the components of system architecture contribute to improving its performance.
2. Explain the circumstances in which a given figure of system performance metric is useful.
3. Explain the usage and inadequacies of benchmarks as a measure of system performance.
4. Describe Amdahl's law and discuss its limitations.
5. Use limit studies or simple calculations to produce order-of-magnitude estimates for a given performance metric in a given context.
6. Use software tools to profile and measure program performance.
7. Design and conduct a performance-oriented experiment of a common system (e.g., an OS and Spark).
8. Conduct a performance experiment on a layered system to determine the effect of a system parameter on system performance.

SF/System Reliability
[1 CS Core hour and 1 KA Core hour]
***Topics**:*

- Distinction between bugs and faults
- Reliability through redundancy: check and retry
- Reliability through redundancy: redundant encoding (error correction codes, CRC, FEC)
- Reliability through redundancy: duplication/mirroring/replicas
- Other approaches to reliability

***Learning Outcomes**:*

1. Explain the distinction between program errors, system errors, and hardware faults (e.g., bad memory) and exceptions (e.g., attempt to divide by zero).
2. Articulate the distinction between detecting, handling, and recovering from faults, and the methods for their implementation.
3. Describe the role of error correction codes in providing error checking and correction techniques in memories, storage, and networks.
4. Apply simple algorithms for exploiting redundant information for the purposes of data correction.
5. Compare different error detection and correction methods for their data overhead, implementation complexity, and relative execution time for encoding, detecting, and correcting errors.

SF/System Security
[1 CS Core hour and 1 KA Core hour]

Topics:

- Common system security issues (e.g., virus, denial-of-service attack and eavesdropping)
- Countermeasures
    - Cryptography
    - Security architecture
    - Intrusion detection systems, firewalls

Learning Outcomes:

1. Describe some common system security issues and give examples.
2. Describe some countermeasures against system security issues.

SF/System Design
[1 CS Core hour and 1 KA Core hour]
Topics:

- Common criteria of system design (e.g., liveness, safety, robustness, scalability and security)
- Designs of representative systems (e.g., Apache web server, Spark and Linux)

Learning Outcomes:

1. Describe common criteria of system design.
2. Given the functionality requirements of a system and its key design criteria, provide a high-level design of this system.
3. Describe the design of some representative systems.

Professional Dispositions

- **Meticulousness:** students must pay attention to details of different perspectives when learning about and evaluating systems.
- **Adaptiveness**: students must be flexible and adaptive when designing systems. Different systems have different requirements, constraints and working scenarios. As such, they require differen designs. Students must be able to make appropriate design decisions correspondingly.

Math Requirements

Required:

- Discrete Math:
    - Sets and relations
    - Basic graph theory
    - Basic logic
- Linear Algebra:
    - Basic matrix operations
- Probability and Statistics
    - Random variable
    - Bayes theorem
    - Expectation and Variation
    - Cumulative distribution function and probability density function

Desirable:

- Basic queueing theory
- Basic stochastic process

Shared Concepts and Crosscutting Themes

Shared Concepts:

- Networking and communication (NC)
- Operating system (OS)
- Architecture and organization (AR)
- Parallel and distributed computing (PDC).

Competency Specifications

- **Task 1:** Describe common criteria of system design
- **Competency Statement:** Given a target scenario (e.g., a client-server system or a database system), describe common criteria to be considered when designing such a system (e.g., liveness, safety, robustness, scalability, and security)
- **Competency area:** Systems
- **Competency unit:** Requirements / Design / Documentation
- **Required knowledge areas and knowledge units:**
    - SF / System Design
    - SF / System Reliability
    - SF / System Security
- **Required skill level:** Explain

- **Core level:**

- **Task 2:** Describe and compare different scheduling algorithms
- **Competency Statement:** Given a common scheduling algorithm (priority-based scheduling, earliest deadline first, and (non)-preemptive scheduling), describe how it works and its pros and cons
- **Competency area:** Systems / Theory
- **Competency unit:** Requirements / Documentation / Evaluation
- **Required knowledge areas and knowledge units:**
    - SF / Resource Allocation and Scheduling
    - SF / System Performance
- **Required skill level:** Explain / Evaluate
- **Core level:**

- **Task 3:** Choose and implement an application-specific scheduling algorithm
- **Competency Statement:** Given a target application scenario, choose an appropriate scheduling algorithm, implement it and evaluate its performance
- **Competency area:** Systems / Application
- **Competency unit:** Requirements / Design / Development / Testing / Documentation / Evaluation
- **Required knowledge areas and knowledge units:**
    - SF / Resource Allocation and Scheduling
    - SF / System Performance
    - SF / Performance Evaluation
    - SF / System Design
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 4:** Design and develop a system
- **Competency Statement:** Given a target scenario (e.g., a client-server application or a database) and requirements (e.g., performance, availability and security), design and develop a system that satisfies the expected requirements.
- **Competency area:** Systems / Application

- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Integration / Documentation / Evaluation / Consumer Acceptance
- **Required knowledge areas and knowledge units:**
    - SF / Basic Concepts
    - SF / System Design
    - SF / System Reliability
    - SF / System Security
    - SF / Performance Evaluation
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 5:** Evaluate the performance of a given system
- **Competency Statement:** Given a system, evaluate its key performance metrics (e.g., correctness, throughput, and average/tail latency) analyticalay and experimentally
- **Competency area:** Systems / Theory
- **Competency unit:** Testing / Deployment / Documentation / Evaluation
- **Required knowledge areas and knowledge units:**
    - SF / System Performance
    - SF / Performance Evaluation
- **Required skill level:** Apply / Evaluate
- **Core level:**

- **Task 6:** Find the performance bottleneck of a given system
- **Competency Statement:** Given a system and its target deployment environment, find its performance bottleneck (e.g., memory, CPU, and networking) through analytical derivation or experimental study
- **Competency area:** Systems / Theory
- **Competency unit:** Design / Testing / Deployment / Documentation / Evaluation
- **Required knowledge areas and knowledge units:**
    - SF / System Performance
    - SF / Performance Evaluation
    - SF / System Design
    - SF / Overview of Computer Systems
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 7:** Find and fix bugs in a system
- **Competency Statement:** Given a system, its deployed environment and its buggy symptoms, find bugs by using tools for program tracing, single stepping, and debugging, and fix them.
- **Competency area:** Software / Systems / Application
- **Competency unit:** Development / Testing / Documentation / Evaluation / Maintenance
- **Required knowledge areas and knowledge units:**
    - SF / System Design
    - SF / System Reliability
    - SF / Performance Evaluation
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

- **Task 8:** Deploy a system in a cloud environment
- **Competency Statement:** Given a system that can run in local machines, deploy it in a cloud environment
- **Competency area:** Software / Systems
- **Competency unit:** Deployment / Integration / Documentation
- **Required knowledge areas and knowledge units:**
    - SF / Overview of Computer Systems
- **Required skill level:** Apply
- **Core level:**

- **Task 9:** Maintaining and evolve a system

- **Competency Statement:** Given a system, maintain its daily operation and evolve it based on customers' feedback
- **Competency area:** Software / Systems
- **Competency unit:** Requirements / Design / Development / Testing / Deployment / Integration / Documentation / Evaluation / Maintenance / Management / Consumer Acceptance / Adaptation to social issues / Improvement
- **Required knowledge areas and knowledge units:**
    - SF / Overview of Computer Systems
    - SF / Basic Concepts
    - SF / System Design
    - SF / System Reliability
    - SF / System Security
    - SF / Performance Evaluation
- **Required skill level:** Apply / Evaluate / Develop
- **Core level:**

Course Packaging Suggestions

Introductory Course to include the following:

- Overview of Computer Systems - 2 hours
- Basic Concepts - 6 hours
- Resource Allocation and Scheduling - 4 hours
- System Performance - 6 hours
- Performance Evaluation - 6 hours
- System Reliability - 4 hours
- System Design - 6 hours
- System Security - 6 hours

Pre-requisites:

- Sets and relations, basic graph theory and basic logic from Discrete Math
- Basic matrix operations from Linear Algebra

- Random variable, Bayes theorem, expectation and variation, cumulative distribution function and probability density function from Probability and Statistics

Skill statement: A student who completes this course should be able to (1) understand the fundamental concepts in computer systems; (2) understand the key design principles, in terms of performance, reliability and security, when designing computer systems; (3) deploy and evaluate representative complex systems (e.g., MySQL and Spark) based on their documentations, and (4) design and implement simple computer systems (e.g., an interactive program, a simple web server, and a simple data storage system).

Advanced Course to include the following:

- System Design - 10 hours
- KUs from OS - 2 hours
- KUs from NC - 2 hours
- KUs from PDC - 2 hours
- KUs from AR - 2 hours
- System Reliability - 10 hours
- System Performance - 6 hours
- System Security - 6 hours

Pre-requisites:

- Basic queueing theory and stochastic process
- Introductory Course of the SF KA

Skill statement: A student who completes this course should be able to (1) have a deeper understanding in the key design principles of computer system design, (2) map such key principles to the designs of classic systems (e.g., Linux, SQL and TCP/IP network stack) as well as that of more recent systems (e.g., Hadoop, Spark and distributed storage systems), and (3) design and implement more complex computer systems (e.g., a file system and a high-performance web server).

Committee

Chair: Qiao Xiang, Xiamen University, Xiamen, China

Members:

- Doug Lea, State University of New York at Oswego, Oswego, NY, USA
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Matthias Hauswirth, University of Lugano, Lugano, Switzerland

- Ennan Zhai, Alibaba Group, Hangzhou, China
- Yutong Liu, Shanghai JiaoTong University, Shanghai, China

Contributors:

- Michael S. Kirkpatrick, James Madison University, Harrisonburg, VA, USA
- Linghe Kong, Shanghai JiaoTong University, Shanghai, China

Appendix: Core Topics and Skill Levels

KA	KU	Topic	Skill	Core	Hours
SF	Overview of Computer Systems	• Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; datapath + control + memory) • Hardware as a computational paradigm: Fundamental logic building blocks; Logic expressions, minimization, sum of product forms • Programming abstractions, interfaces, use of libraries • Distinction between application and OS services, remote procedure call • Application-OS interaction • Basic concept of pipelining, overlapped processing stages • Basic concept of scaling: going faster vs. handling larger problems	Explain	CS	3
SF	Basic Concepts	• Digital vs. Analog/Discrete vs. Continuous Systems • Simple logic gates, logical expressions, Boolean logic simplification • Clocks, State, Sequencing • State and state transition (e.g., starting state, final state, life cycle of states)	Apply	CS	4

		● Finite state machines (e.g., NFA, DFA) ● Combinational Logic, Sequential Logic, Registers, Memories ● Computers and Network Protocols as examples of State Machines			
SF	Resource Allocation and Scheduling	● Different types of resources (e.g., processor share, memory, disk, net bandwidth) ● Common scheduling algorithms (e.g., first-come-first-serve scheduling, priority-based scheduling, fair scheduling and preemptive scheduling) ● Advantages and disadvantages of common scheduling algorithms	Explain	CS/KA	1/2

SF	System Performance	● Latencies in computer systems
		  ○ Speed of light and computers (one foot per nanosecond vs. one GHz clocks)
		  ○ Memory vs. disk latencies vs. across the network memory
		● Caches and the effects of spatial and temporal locality on performance in processors and systems
		● Caches and cache coherency in databases, operating systems, distributed systems, and computer architecture
		● Introduction into the processor memory hierarchy and the formula for average memory access time
		● Rationale of virtualization and isolation: protection and predictable performance
		● Levels of indirection, illustrated by virtual memory for managing physical memory resources
		● Methods for implementing virtual memory and virtual machines
			Apply	CS/KA	2/2

SF	Performance Evaluation	● Performance figures of merit	Evaluate	CS/KA	2/2

		● Workloads and representative benchmarks, and methods of collecting and analyzing performance figures of merit ● CPI (Cycles per Instruction) equation as tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations. ● Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can ● Analytical tools to guide quantitative evaluation ● Order of magnitude analysis (Big O notation) ● Analysis of slow and fast paths of a system ● Events on their effect on performance (e.g., instruction stalls, cache misses, page faults) ● Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation ● Microbenchmarking pitfalls			
SF	System Reliability	● Distinction between bugs and faults ● Reliability through redundancy: check and retry ● Reliability through redundancy: redundant encoding (error correction codes, CRC, FEC) ● Reliability through redundancy: duplication/mirroring/replicas ● Other approaches to reliability	Evaluate	CS/KA	2/1
SF	System Security	● Common system security issues (e.g., virus, denial-of-service attack and eavesdropping) ● Countermeasures ○ Cryptography	Evaluate	CS/KA	2/1

		○ Security architecture ○ Intrusion detection systems, firewalls			
SF	**System Design**	● Common criteria of system design (e.g., liveness, safety, robustness, scalability and security) ● Designs of representative systems (e.g., Apache web server, Spark and Linux)	Design	CS/KA	2/1

Course Packaging by Competency Area

- **Software:** Courses that span Software Development Fundamentals (SDF), Algorithms and Complexity (AL), Programming Languages (PL) and Software Engineering (SE).

- **Systems:** Courses that span Systems Fundamentals (SF), Architecture and Organization (AR), Operating Systems (OS), Parallel and Distributed Computing (PDC), Networking and Communication (NC), Security (SEC) and Data Management (DM).

- **Applications:** Courses span Graphics and Interactive Techniques (GIT), Artificial Intelligence (AI), Specialized Platform Development (SPD), Human-Computer Interaction (HCI), Security (SEC) and Data Management (DM).

Curricular Packaging Suggestions

- For curricula that focus on **Software**

- For curricula that focus on **Systems**

- For curricula that focus on **Applications**

Competency Specifications for Competency Areas

- **Software:** Specifications that span Software Development Fundamentals (SDF), Algorithms and Complexity (AL), Programming Languages (PL) and Software Engineering (SE).

- **Systems:** Specifications that span Systems Fundamentals (SF), Architecture and Organization (AR), Operating Systems (OS), Parallel and Distributed Computing (PDC), Networking and Communication (NC), Security (SEC) and Data Management (DM).

- **Applications:** Specifications span Graphics and Interactive Techniques (GIT), Artificial Intelligence (AI), Specialized Platform Development (SPD), Human-Computer Interaction (HCI), Security (SEC) and Data Management (DM).

Curricular Practices

- **Social aspects**
  - Teaching about Accessibility in CS education
  - Computing for Social Good – Education
  - Ethical Practices in Global CS Education: Perspectives from the Souths
  - Ethics in CS education
- **Professional practices**
  - CS education in the liberal arts context
  - CS education in community colleges
  - CS education in China
  - CS education in the Arab World
  - CS education in Latin America
  - CS education in Africa
- **Programmatic considerations**
  - CS + X
  - Future of CS educational materials
  - Connecting concepts across knowledge areas