**

**ECE 4960: Scientific and Numerical Computation**

**Spring 2019**

**Week 1: Class Introduction**

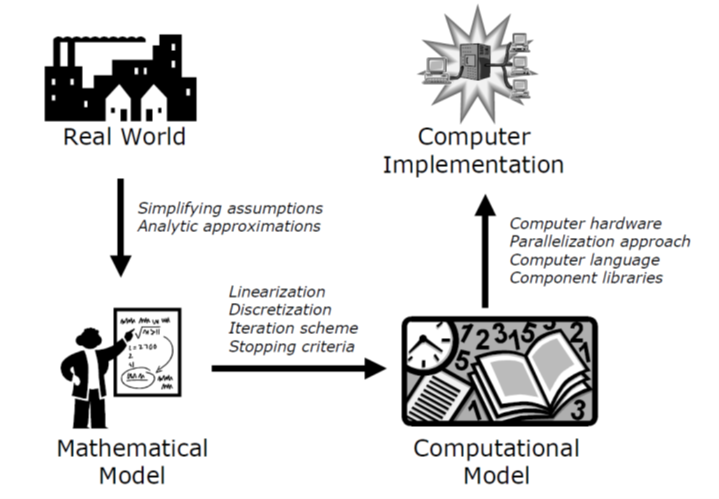
**Reading Assignments:**

1. Chap. 1, D. Bindel and J. Goodman, *Principles of Scientific Computing*, 2009.
2. Chaps. 2 and 8, B. Einarsson, Ed., *Accuracy and Reliability in Scientific Computing*, SIAM 2005.
3. (Optional) Chaps. 6 – 8, S. Oliveira and D. Stewart, *Writing Scientific Software: A Guide to Good Style*, Cambridge 2006.

**1.1 ECE 4960 is:**

* A senior/first-year graduate level of programming class that teaches you how to write **robust** and **verifiable** software to interface with the **physical world** described by mathematical laws. The implementation cycle is shown in Fig. 1.

**Fig. 1.** Programming to interface or to emulate the real world from the mathematical modeling. Both computational and software perspectives are essential.



* When you write software for an arbitrary problem (such as a computer game or word processing), the first thing you need in the top-down design is the **specification**. Writing complete and consistent specification leads to the overall object structures and inter-object application procedural interface (API), which takes years of software development experience.
* Inconsistent specification is often the source of many bugs later on in the software lifecycle.
* For this class, we jump over most of the “specification” problem for writing software on the physical world, which is naturally described by the physical laws.
* The physical world is most often harmonious, and contains many different feedback paths to regulate abnormal situations.
* The physical world, and hence the software that describes it, has nearly NO “inconsistency” problem. One of the ways for artificial problems to avoid inconsistent specification is to mimic the physical behavior in the nature.
* The main focus of this class is on writing software to model the scientific and engineering problems, as well as problems derived from virtual reality (VR) and augmented reality (AR). Most of these problems are still too large to ignore the hardware implementation, such as circuit simulation of a chip, weather simulation, 3D image/geometry processing, etc.
* We will differentiate between **hardware-aware** programming and **fast-prototype** programming, where execution efficiency tradeoffs with *programmer productivity* and *platform tolerance*, as shown in Fig. 2. For example, for most assignments, you will need to be able to estimate the computational efforts and memory usage instead of assuming the computational resources are infinite.
* When you start learning programming, your instructor will give you a specific problem to solve, as well as a test or a demonstration to give. You program to give the right answer of the test or to achieve the demonstration goal, sometime with run time and memory benchmarks. Your instructor will give you grade based on your results, and sometime comment on your programming styles
* Role reversal on assessment in this class: In commercial software, developers, instead of users, provide specification, validation, and test cases. Likewise you will assume similar responsibilities in defining how to assess your results and prove that your code is correct, instead of just fulfilling given specification and test suites.
* Overspecification or oversupervision during the training phase results in immature learners as well as programmers, as shown in Fig. 3 for a Chicago mathematical education assessment. This is often true for high school students, as well as your becoming a mature programmer.
* Developers should also be in the position of novel users and attackers! Robustness **only** comes from comprehensive testing in all levels from the **user** perspective. As was found for thousand years in war strategy: “There is no guaranteed victory, but there are careful ways to avoid failure!”

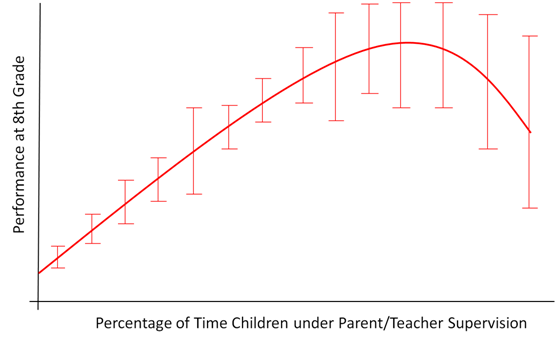
**Programmer productivity**

**Platform independence**

**Runtime and memory efficiency**

**System controllability**

**Fig. 2.** Programming abstraction level vs. System performance and controllability.



**Fig. 3.** The philosophy behind self assessment of program assignment. The parenting curve is drawn from: Stephen J. Dubner and Steven Levitt, *Freakonomics: A Rogue Economist Explores the Hidden Side of Everything, Chap. 5*, William Morrow, 2005.

**Group discussion**: How to make it fair to prescribe your own testing methods for class assignments?

**1.2 ECE 4960 is not:**

* This is not a class to teach fundamental program skills: You are expected to have substantial programming experience in a high-level, **object-oriented** language. Although you are allowed to use any language and developer platform of your choice, Gnu C++ will be most often used as programming practice examples. Programming techniques such as sorting and tree trasversing are taught in basic data structure classes, and are implicitly assumed that you know those subjects well already.
* This is not a mathematical class to prove convergence or error bounds, although concepts for testing and verification will be reviewed when needed.
* This is not a class to focus on Internet or Cloud computing environment where programmer productivity and machine compatibility from virtual wrappers are emphasized. Run time and memory efficiency are frequently part of the goals in your program assignments.
* This is not an operating system or compiler class to deal with system setup and resource allocation, although you will gain some practical knowledge of the system. Detailed knowledge of compilers and operating systems belongs to other courses.

**1.3 Syllabus:**

* Sign up for Blackboard access: ECE 4960 Special Topics.
* Lectures TTh 10:10am – 11:25am in Phillips 407. I will not do role call, and attendance is not explicitly considered in grading. However, there will be “real-time hacking assignments” to enhance your programming ability. An announcement will be posted on Blackboard for you to finish the assignment within a given time, but you will miss the “speed bonus” if there is one and you are not present in class.
* Recitation/Lab: Wednesday: 7:30pm – 9pm in Philips 403. The are two main functions: introduction of good large-scale programming practice and project practice. This will also be the time reserved for make-up lectures if necessary.

**Group discussion**: What is the most important ONE thing you hope to learn in this class?

**2. Software development overview**

**2.1 The lifecycle of programming**

* Three things for quality assurance: Testing, testing, and testing. Quality assurance means that sufficient tests have been done for verification.
* Three things for longetivity and re-use of codes:
  1. Clear specification of API and object models;
  2. Evolution of source code together with regressional tests;
  3. In-code and external documentation
* In any software that is jointly developed by a team, the source code is literally “holy”. Separate out your random tests and execute them accordingly with the code evolution.
* Make your code readable by following coding convention.

**2.2 Source of errors in software**

Murphy’s Law: “Anything that can go wrong, will go wrong.”

Yhprum’s Law: “Anything that can go wrong, will go right.” (Or Chinese saying: “The path will become straight after you make the turn.” “The boat can go straight after passing the bridge.”

Murphy’s Law for things with complexity: “If one thing goes wrong, everything else will, and at the same time”

Bugs, bugs, bugs:

* Specification errors
* Logic errors
* Coding errors
* Approximation and convergence errors
* Conversion errors
* Memory errors
* Memory and resource leaks
* Multi-threaded or race errors
* Timing errors
* Distributed application errors
* Storage errors
* Procedure integration errors
* Version errors and backward compatibility

**Discussion:** What is your most vivid experience in programming errors?