Advanced Programming in the Mechanical Engineering Curriculum

B.D. Coller
Department of Mechanical Engineering
Northern Illinois University
DeKalb, Illinois 60115

1. Introduction

We are in the process of developing an advanced computing and programming track within the undergraduate mechanical engineering curriculum at Northern Illinois University (NIU). We are introducing our mechanical engineering students to concepts such as object oriented programming, data structures, complexity analysis, and elements of software design that are normally taught to computer scientists. Rather than ship our engineering students to the computer science department, we provide an authentic engineering context, designed to engage students, in which to learn the material.

On its surface, the context, looks like a multi-player video game. A screen shot of the game is shown in Figure 1. Deep inside, however, it is a sophisticated automobile simulation that the students must write much of themselves over a sequence of several courses. Here we aim to leverage the tremendous popularity of video games with this generation of students, and direct their enthusiasm toward educational purposes.

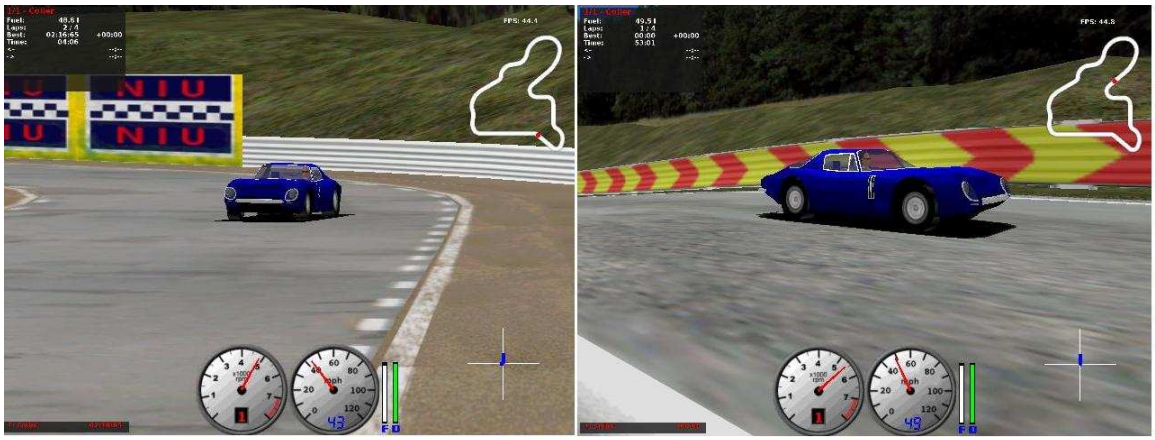


Figure 1 Snapshots of NIU-TORCS, the primary software package.

In this paper, we outline the advanced computing track being developed at NIU, providing an outline of the courses and topics we will teach, and describing the hardware and software infrastructure necessary to support the endeavor. First, however, we discuss our motivation for the project.

2. Motivation

Formal instruction that our undergraduate engineering students receive in computer programming is similar to that experienced by undergraduates throughout the country; it has changed little over the past several decades. Whether in Fortran, Pascal, Java, Matlab, or C/C++, students learn in their introductory programming course the basics of how to piece together flow control structures, and input/output capabilities. Students become proficient at writing computer programs that tend to be small in scale and narrow in scope. While this is a fine place to start, we are dismayed by the fact that this is also the place where most engineering curricula stop teaching their students programming.

Engineering students typically do get at least one more opportunity to write computer programs: in their numerical methods course. However, if the top selling text books are any indication of what happens in these courses, very little energy or effort is devoted to program design, programming style/technique, computational efficiency or scalability. Top selling numerical methods textbooks are catalogs of techniques presented generically (independent of any computational platform), followed by a bare-minimum series of commands or computer code snippets that will implement the recipe just described in a variety of different software packages or programming languages. Whether consciously or unconsciously, the text authors and course instructors are grooming the students to be able to solve the types of problems one finds at the end of the chapter: small in scope, narrow in focus, again.

The approach might be well suited for the 1960's and 1970's, when computing and programming became a core component of the undergraduate engineering curriculum, and when computing technology severely limited the size and scope of even the most ambitious computational projects. The computational landscape we see before us now, though, is dramatically different. The computational tools that engineers use on a daily bases are typically parts huge software packages written by armies of programmers, computer scientists, artists, and engineers. Unfortunately, the computing/programming skills that we teach our engineering students do not scale well. Programming "in-the-large" requires different sets of skills than those they learn in their introductory courses. The danger is that as the gap between the state-of-the-art computational technology and what engineers know grows wider, engineers will play a smaller role in the development of software that is such a critical component of modern engineering practice.

In developing our computational track for undergraduate mechanical engineering students at NIU, we are NOT attempting to train our nascent engineers as computer scients. Instead, our goal is to make them better engineers. Our coverage of computing and programming topics is not as rigorous as one would normally find in the computer science curriculum. Instead we hope that an introduction to the topics will open new vistas on how to solve engineering problems – both analysis and design.

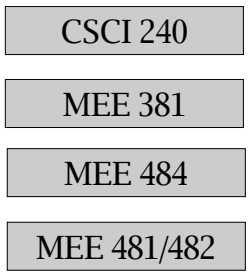
Finally, one of the most important features of the curriculum track, we believe, is that the bulk of it takes place within mechanical engineering itself. Our students pursue their

mechanical engineering degrees because they are interested in things mechanical: cars, robots, spacecraft, airplanes, cranes, trains, et cetera. Nonetheless, our students who take additional courses, on their own initiative, from the Computer Science typically learn by working on applications in data bases, information management, accounting, and other business related areas. Students must make the appropriate connections back to engineering themselves, a difficult task for the novice. By grounding the computational topics in mechanical engineering we are better able to motivate our students and better able to demonstrate the power in these tools. The problem-based learning paradigm has a proven track record [3,4,5,9].

3. The Advanced Computing Track

As indicated in the introduction, there is a single project that serves as an anchor for the advanced computing track. It is a vehicle simulation that looks and acts like a modern video game. At it its heart, though, is a sophisticated simulation of a complex engineering system. The project defines the skills we want our students to learn, and it provides the context that gives extra meaning to the learning experience.

The advanced computing curriculum track at NIU consists of five courses as illistrated to the right, for a total of 15 credit hours. The first is a standard introduction to programming course offered by the Computer Science Department that all engineering students must take. Beyond that is a series of mechanical engineering courses, each of which takes on a self-contained portion of the video game/simulation master project.



3.1 CSCI 240. Introduction to Computer Programming in C++ (4 credits)

As just stated, this is the standard introductory computer programming course that all engineering students are required to take. Although the title lists C++ as the programming language, the first 14 out of 15 weeks focus on the traditional (C-like) procedural programming aspects of the language. In the final week, students begin learning what a class is.

3.2 MEE 381. Numerical Methods and Programming in Engineering Design (3 credits)

For students participating in the advanced computing track, this course replaces, MEE 380, the traditional numerical methods course. It is being offered for the first time in Spring of 2005. The course is organized around a single large project: write a computer code for the car to drive itself around the track as fast as possible. The code that the students write will be integrated into the larger game/simulation software. At the end of the semester, all students compete in a friendly head to head race.

Through the application programming interface (API), students have direct access to relatively few dynamic variables and parameters of the car. These include distance of the car from the center of the track, distance from the start line, and orientation of the car relative to the track among others. From these, their programs must calculate the optimal times to shift gears, how fast one may drive around a given corner and still maintain control, when to accelerate when coming out of a turn, when to begin braking prior to entering a turn, and much more. These tasks require students to perform curve fitting, root finding, solutions of simultaneous algebraic equations, numerical differentiation, numerical integration.

Much more than a data extrapolation exercise, the project offers ample room for creativity and inventiveness. Students must devise strategies for passing their opponents' cars, and for making it more difficult for their opponents to pass them. They must choose a path on the track that best suits the characteristics of their car

Figure 2

In addition to the numerical methods, the project provides a natural context for learning object-based programming in C++. Students create classes for the driver of their own car, classes for their opponents' cars, classes for speedometers, accelerometers, anti-lock braking system, traction control system, and track. Through direct experience, they appreciate the benefits of encapsulation. Additionally, they employ other features of the C++ language beneficial to scientific computing such as inline functions and function overloading. Furthermore students are introduced to profiling, complexity analysis, and using XML for specifying simulation parameters.

To make room for the programming topics, we do not cover numerical methods in MEE 381 to the same breadth as is done in a more traditional course. For example, in discussing solutions to linear algebraic equations, we do not cover the broad spectrum of algorithms for structured and unstructured matrices. Instead, we discuss two that work for our problems, one which is significantly faster than the other. We discuss their features, and then briefly mention that there are other more appropriate recipes for different situations.

3.3 MEE 484. Advanced Computing in Mechanical Engineering (3 credits)

The next course in the sequence, MEE 484, is a technical elective to be offered for the first time in Fall 2005. In it, students form teams to write the bulk of the code for

simulating the car itself. The car is an ideal setting to learn object oriented programming. It is straightforward to define the software objects: they correspond to the physical components of the car: its engine, transmission, differential, tires, steer mechanism, suspension, and more. Students learn about inheritance and polymorphism in C++. They are introduced to elementary data structures: linked lists, trees. We discover the Standard Template Library. To facilitate their team software development, the students use version control, and learn elements of the Unified Modeling Language (UML).

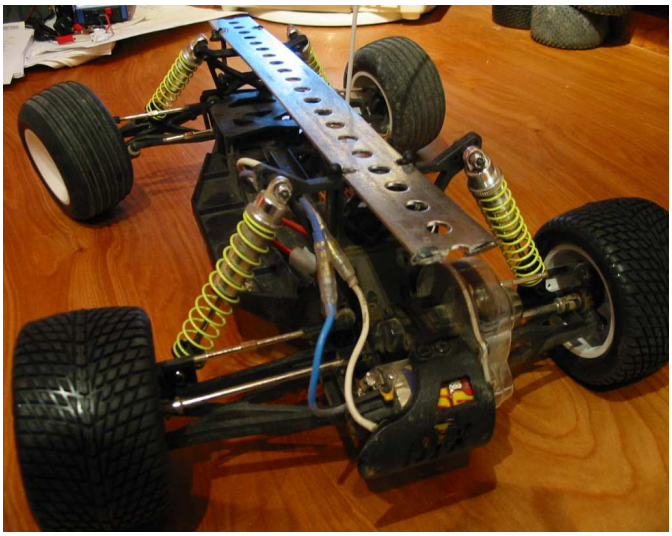


Figure 3 Radio controlled car used for validation.

Meanwhile, all but the greasiest of gear-heads learn a significant amount about the engineering of automotive systems as well. All teams of students will be given the same task that they must complete over the duration of the semester: to develop a computational model for the radio-controlled car shown in Figure 3. The radio-controlled car shown in the picture is a toy, but it is a elaborate toy. It has a full independent wishbone suspension for which it is possible to change linkage geometry, spring stiffnesses, and damping characteristics. Mass properties are easily adjusted. By choosing the radio-controlled car as our target system to model, we can perform one of the most important steps in computational modeling: validation. Here, we are mostly concerned that the simulation correctly predicts the qualitative effects: for example, how mass distribution affects over-steer or under-steer of the vehicle. Both the physical system and the simulation should show the same behavior.

3.4 MEE 481/482: Senior Design (5 credits total)

The final step in the advanced computing curriculum track is the integration of the vehicle simulation project into the capstone senior design project. At NIU, this is a two semester sequence of courses. The first, MEE 481, is a design seminar in which students form teams, and formulate a design project. Typically students participating in the curriculum track will take MEE 481 and MEE 484 concurrently, so they learn about the design issues as they are learning about the car and capabilities and limitations of the computational model of the car.

In the second semester, MEE 482, the students design. They perform a systems-level design of the car with the goal of creating the fastest, most agile car possible within the constraints imposed upon them. There are dozens of design parameters that the students must choose, including car geometry (wheel base, track, center of mass location),

suspension geometry, suspension stiffnesses, damping, steering geometry, brake disk and caliper sizes, differential characteristics, and transmission gear ratios among others. Students must choose among a selection of engines and tires. Students will have the freedom to explore more advanced strategies which distribute braking/tractive effort over the four wheels independently, depending on the dynamic state of the car. In the race at the end of the final semester, students drive their own cars for at least one of the events. Therefore, they need to design a graphical user interface which brings all the necessary information to the driver without distracting his/her attention from driving.

Because of the expense of building prototypes, companies would like to shift as much of the engineering design process to the computational domain as possible. While it is commonplace to design an individual part with finite element software, complex interconnected systems such as the automobile are orders of magnitude more difficult. After constructing a reasonable computational model of the physical system, one must properly interpret and make physical sense of the torrent of numbers that come gushing out of the simulations. One must be able to use the data to then navigate large design spaces, and finally arrive at a good result. We give our senior-level engineering students precisely this type of design experience, an experience that is normally very difficult to incorporate in the curriculum.

4. Infrastructure for the Curriculum Track

Below, we list the software and and computer hardware that one needs to implement the advanced computing track. As you shall see, it requires a minimal investment since we use common commodity off-the-shelf equipment, and most of our software is free.

4.1 Primary Software

The primary software package we use for the course sequence is NIU-Torcs. It is based on a program called TORCS (The Open Racing Cars Simulator, www.torcs.org), written by Eric Espie, Christophe Guionneau, Bernhard Wymann, Christos Dimitrakakis, and others. The original TORCS program was designed as a video game. It is available under the GNU General Public License (GPL), which means that its source code is available with the distribution. We borrow heavily from it, particularly the 3-D graphics components. Other portions, including the race engine components, the simulation module, and interfaces have been extensively modified or rewritten to suit our educational purposes.

TORCS is available free of charge. Likewise, our derived program NIU-TORCS will be made available free of charge, along with the source code, once we have had ample opportunity to work the majority bugs out.

4.2 Computing Platform

TORCS runs on PCs with the Linux operating system and on Microsoft Windows. It is

developed primarily on the Linux platform; Windows compatibility (tied to the Visual C++ compiler) gets added later. We follow the same pattern in our development of NIU-TORCS. As we write this, NIU-TORCS only runs in Linux. If funding persists, or if someone volunteers to do much of the work, we would we would consider porting the project to Windows sooner rather than later.

Nonetheless, we would argue that Linux is the ideal computing platform for the educational project. We'll spare the reader a diatribe on how Linux is more stable, more secure, and more efficient than Windows. We will simply mention that the operating system is free, and all the programming/software development tools we use in the Linux environment are of extremely high quality and free as well.

The only cost is that of the computers themselves. We are using common Pentium 4, 3.2 GHz computers with 512 Mb of RAM, each of which cost less than $900 (without monitor) in December of 2004. However, our computers are much more powerful than necessary. As a general rule of thumb, if the computer has sufficient power to run Windows XP, it has ample power to run Linux and NIU-TORCS as well. The only caveat is that the computer needs a 3D graphics card with OpenGL support. One can purchase a sufficient graphics card for less than $70.

4.3 Ancillary Software and Equipment

In order for students to evaluate the performance of their code, they need a means of plotting data that their simulations generate. We use Matlab; we already have a license for the student computer labs. A free alternative would be to use Octave and/or Gnuplot.

Figure 4 Matlab plotting tool for analyzing performance.

In race events that students (rather than computer algorithms) drive the virtual cars, we need appropriate input devices. It is possible to drive a car using the keyboard or mouse. However, these are clumsy. Instead, we use steering wheels with pedals, that one can purchase for roughly $40 almost anywhere PC video games are sold. The steering wheels plug into the computer's USB port. In the Fedora Core 2 Linux distribution, there is built-in support for steering wheels and other types of joysticks that are human

interface device (hid) compliant. There is no support for force feedback steering wheels yet. TORCS provides a built in tool for calibrating joysticks/steering wheels.

5. Closing

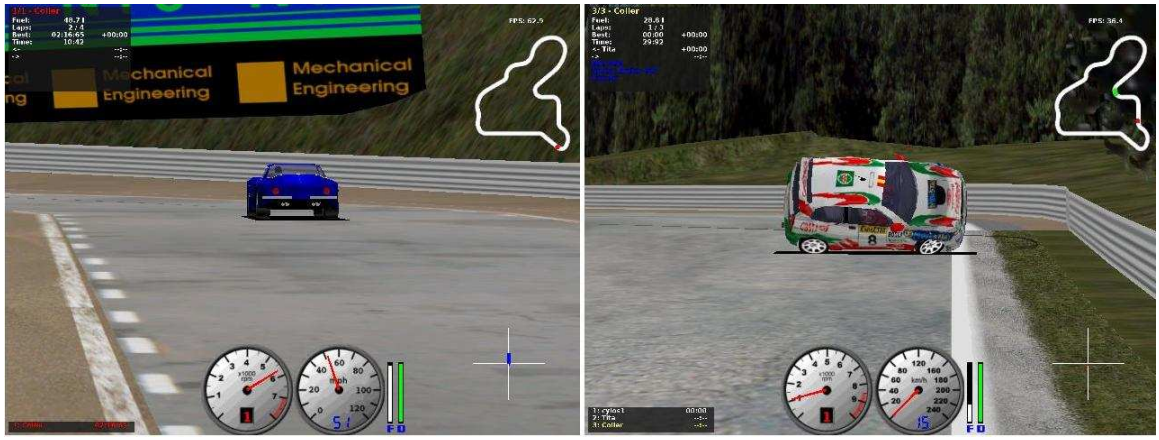
In a recent article on the role of computing in education, G.V. Wilson writes [8]:

> Good computing practice is just as important to physical scientists and engineers today as good laboratory practice and sound mathematics. My experience has been that it takes a few months to teach a physicist, geologist, or biochemist enough to make a big difference in her productivity. Sadly, physical scientists who want to learn such things usually have to teach themselves.

The project described here provides an authentic context in which our undergraduate mechanical engineers gain such computing expertise and more. Programming, design, simulation, and analysis are all intertwined in a project-based setting designed to be engaging for engineering students. We have patterned our approach after what cognitive psychologists call a macro-context [7], connecting lessons and assignments to a meaningful overall goal that gives learning purpose. Research has shown that such contextualized learning is significantly more effective than traditional classroom learning [1,2,6].

In addition to enriching the educational experiences of students at NIU, we wish to make a broader impact. We plan to make the core software and course materials available to educators everywhere. These items will be made available after we have had the chance to go through the course cycle at least once and make refinements.

In addition, we hope to integrate the computing and simulation framework into other parts of the mechanical engineering curriculum. The most natural places to focus on are the undergraduate and graduate control classes we teach.



Acknowledgment

The author gratefully acknowledges support from the National Science Foundation under grant 0354557. Any opinions, findings, and conclusions are those of the author and do not necessarily reflect those of the National Science Foundation.

References

[1] J. Bransford, T. Hasselbring, B. Barron, S. Kulewicz, J. Littlefield, and L. Goin. Uses of macro-contexts to facilitate mathematical thinking. In T*he teaching and Assessing of Mathematical Problem Solving*, pages 125—147. Lawrence Erlbaum Associates, 1988.

[2] J. Bransford, N. Vye, C. Kinzer, and V. Risko. Teaching thinking and content knowledge: Toward an integrated approach. In B. Jones and L. Idol, editors, *Dimensions of thinking and cognitive instruction*, pages 381—413. Lawrence Erbaum Associates, 1990.

[3] D. Johnson, R. Johnson, and K. Smith. *Active Learning: Cooperation in the College Classroom*. Interactive Book Company.

[4] S.D. Sheppard and R. Jenison. Freshmen engineering design experiences: an organizational framework. *International Journal of Engineering Education*, 13, 1997.

[5] S.D. Sheppard, R. Jenison, A. Agogino, M. Brereton, L. Bucciarelli, J. Dally, J. Demel, C. Dym, D. Evans, R. Faste, M. Henderson, P. Minderman, J. Mitchell, A. Oladipupo, M. Piket-May, R. Quinn, T. Regan, and J. Wujek. Examples of freshman design education. *International Journal of Engineering Education*, 13, 1997.

[6] R. Sherwood, C. Kinzer, T. Hasselbring, and J. Bransford. Macro-contexts for learning: Initial findings and issues. Journal of Applied Cognition, 1:93—108, 1987.

[7] J. Van Haneghan, L. Barron, M. Young, S. Williams, N. Vye, and J. Bransford. The Jasper Series: An experiment with new ways to enhance mathematical thinking. In *Enhancing Thinking Skills in the Sciences and Mathematics*, pages 15—38. Lawrence Erlbaum Associates, 1992.

[8] G.V. Wilson. Teaching horses to wistle: An apostate's view of parallel computing and the undergraduate computational science curriculum. In *Forum on Parallel Computing Curricula*, 1995.

[9] D. Woods. Problem-Based Learning: How to Gain the Most from PBL. McMaster University. Order from (905) 525-9140.