Implementing Computational Methods into Classes throughout the Undergraduate Chemical Engineering Curriculum

William B. Perry, Victor H. Barocas, and David E. Clough
University of Colorado

Abstract

In previous years, the undergraduate Chemical Engineering curriculum at the University of Colorado has contained a gap in students' exposure to computational methods and programming. As freshmen, students learned programming concepts in the course *Introduction to Engineering Computing* (GEEN 1300) and were later required to use these skills as seniors in *Numerical Methods for Process Simulation* (CHEN 4580). In the two years separating these classes, students had little opportunity to use and reinforce their programming skills. To remedy this oversight, we have developed programming modules for six sophomore- and junior-level courses throughout the curriculum. These modules have been implemented in the courses as supplements to homework.

Each module focuses on a problem that is familiar to students from course material. Students are given a sample program that uses computational methods to solve each problem. They are then asked to modify the program to solve a more difficult problem. In addition to writing these modules, we have also provided support for students in the form of "Module Teaching Assistants." Initially, these modules have proven successful in giving students exposure to programming. The need for steady reinforcement of computational skills is not restricted to Chemical Engineering. The programming module concept would be applicable to any engineering curriculum.

Introduction

The undergraduate Chemical Engineering curriculum at the University of Colorado currently requires students to take two computer-intensive courses. Entering freshmen are required to take *Introduction to Engineering Computing* (GEEN 1300). In this course, students are introduced to computational methods and become familiar with programming principles, various pieces of software, and the computing networks at the university. Currently, the Chemical Engineering section of this course, taught by David Clough, teaches programming skills using Fortran 90. The course also introduces students to Microsoft Excel, Matlab, and Mathcad. In the fall semester of the senior year, undergraduate students are required to take *Numerical Methods for Process Simulation* (CHEN 4580). In this course, taught by Victor Barocas, students are introduced to useful methods for numerical problem solving, such as linear and nonlinear equation solvers, numerical integration, and ordinary differential equation solvers. Although CHEN 4580 students are given the option to use any programming language for homework assignments and projects, most find the Matlab platform to be extremely useful.

Unfortunately, these two computing courses are separated by five semesters in the current curriculum. In the first course, students learn programming concepts and are then expected to remember and use these skills in the second course. In the two years separating these courses, the average student has little opportunity to use his/her programming skills in the normal Chemical Engineering courses. Without use, these skills fade and must be relearned in the later course. Maintaining the programming skills learned in GEEN 1300 is not only important for success in CHEN 4580, but also for engineering problem solving in general.

To correct this problem, we have incorporated programming modules into six Chemical Engineering courses in between GEEN 1300 and CHEN 4580. Each module introduces a problem relating to material in the course, which must be solved using programming and computational methods. Listed below are the six sophomore- and junior-level courses in which programming modules have been implemented:

Year	**Semester**	**Course(s)**
Sophomore	Fall	CHEN 2120 *Material and Energy Balances*
Sophomore	Spring	CHEN 3200 *Chemical Engineering Principles 1 (Fluid Mechanics)*
Junior	Fall	CHEN 3210 *Chemical Engineering Principles 2 (Heat Transfer)* CHEN 3838 *Applied Data Analysis*
Junior	Spring	CHEN 3220 *Chemical Engineering Principles 3 (Separations and Mass Transfer)* CHEN 3320 *Thermodynamics*

Cases in which students may take these classes in a different order are rare, but do exist. The ordering of the modules is important because the problems are intended to increase in difficulty and complement the techniques used in other modules. However, the order in which the modules are worked is not critical and it is far more important that students have the opportunity to work every module regardless of order.

Each module focuses on a target computational technique to be used in solving the problem. The basic module outline consists of background information on this target computational technique followed by the actual problem statements. Each module contains at least two major programming problems. First, the module introduces a basic "tutorial" problem and gives

students a complete program with which to solve it. Students must then type in and run this program. Next, the module gives one or more "challenge" problems, which are considerably more difficult. Students are then asked to modify the previous program to solve the new problem(s). In this way, the module does not ask students to recall all the syntax and structure of a programming language. Giving students a functional program to solve the tutorial problem allows them to refresh their memories on how the language works so that they are never asked to write a program from scratch. Although it is important that students be able to write a complete program on their own, the primary purpose of the modules is to reinforce students' programming skills and help them remain comfortable with programming.

Generally, the tutorial problem is simple enough to have an analytical solution. Students are then asked to compare this solution to the program's numerical solution. In contrast, the challenge problem does not have an analytical solution and can *only* be solved numerically. This type of structure illustrates to students that a pen, paper, and calculator can only take them so far. At some point, the level of difficulty of the problem warrants the use of a computer.

In the courses, these modules were implemented as full homework assignments, with length comparable to that of other homework assignments in the course. The modules in the fall semester ranged from 9-18 pages long and took students anywhere from 5-10 hours to complete. All six modules are available for use on the Computational Methods in Chemical Engineering web site at http://casablanca.colorado.edu/~barocas/Computational.html.

Module Problem Descriptions

Developing the module problems is a challenging task because the problems must blend a series of computational methods problems with course topics that follow the Chemical Engineering curriculum. Making this task even more difficult is the fact that the computational methods should be introduced in order of difficulty, with each problem ideally being progressively more challenging in its programming requirements. This section details each of the six module problems, describing both the computational methods and the course topics we have chosen for the problems.

The first step in developing the module problems is to consider students' understanding of both material in the course and their programming backgrounds. In terms of their experience in computational methods, GEEN 1300 students will have worked extensively with at least one of the following equation solvers: the bisection method, the false position method, or Newton's methods. Through homework assignments, students may also have used Euler's Method, Simpson's rule, and the trapezoidal method for numerical integration in addition to the Golden Section Search Method for optimization.

Considering this background, students in the Material and Energy Balances course should be able to handle a problem with significant computational methods requirements. The tutorial problem for this module uses the Golden Section Search Method to maximize the profit of an isothermal reaction. At constant temperature, only mass balances are required in solving this

problem. The challenge problem then asks students to consider an exothermic reaction and again maximize the profit. For this challenge problem students must add energy balances to their list of equations. Optimization problems fit very well in Material and Energy Balances and can serve as a cumulative review of topics covered throughout the semester.

In the following semester, students in the Fluids course are introduced to numerical solutions of ordinary differential equations (ODEs) by the finite-difference method. The module involves Poiseuille flow of Newtonian (tutorial problem) and non-Newtonian (challenge problem) flow in a pipe. The concept of non-Newtonian fluids is introduced in the course, but up until now the students have had no quantitative experience with such fluids. The module thus allows students to improve both their understanding of fluid mechanics and their computer skills.

In the fall semester of the junior year students have programming modules in two courses. In the Heat Transfer course, students again work with ODE solvers, but in a more difficult problem. Here, the tutorial problem asks students to determine the temperature profile of the familiar cooling fin. This problem generates a second-order ODE, which can be solved both analytically and numerically. In order to aid students with the numerical solution, they are given a first-order ODE solver routine. However, because the problem statement gives the temperature at the base of the fin and a flux boundary condition at the tip of the fin, the numerical solution requires students to implement a shooting method. Effectively, students must combine the ODE solver from the tutorial problem with an equation solver. Once the shooting method algorithm is complete, students are asked to again use it to solve the cooling fin problem—but this time they are asked to consider both convection *and* radiation. This module complements the finite difference methods used for a second-order ODE in the fluids module.

In the other fall junior-level course, Applied Data Analysis, students are asked to fit a data set to a two-parameter linear model. The analytical solution to this problem requires use of the least-squares estimator equations. To determine a numerical solution, students are given a program that minimizes the sum of the squares of the errors (SSE), using the method of steepest descent (the gradient method). Here, the analytical solution serves, not only as a basis of comparison, but also as an initial guess for the two model parameters. For the challenge problem, students are asked to improve the algorithm and use it to fit the data to a nonlinear model.

In the Spring Junior semester, students again have programming modules in two of their courses. In the Mass Transfer and Separations course, students use Newton-Raphson iteration with numerical differentiation to determine the mass balance on an equilibrium flash (tutorial problem) and then on two flashes (challenge problem) in series for an azeotropic system. Again, students develop both computational and classical skills. For the Thermodynamics course a modified line search is used to generate a ternary phase diagram. Liquid phase behavior is an important component of thermodynamics, and it is also important that students recognize that bracketing methods, although less efficient locally than Newton, can be of great value in finite-domain problems (note that Newton's method can struggle in such problems because it allows mole fractions outside of [0,1]).

By the time students reach CHEN 4580, they will be comfortable writing programs and will have used several computational techniques.

Software

Because the initial purpose of creating the programming modules was to bridge GEEN 1300 and CHEN 4580, the software used in both of these courses must be considered when selecting the programming software for the modules.

After completing GEEN 1300, students should feel very comfortable writing a Fortran 90 program and will have an excellent understanding of the language. Fortran 90 is used for this course because many engineers will find it useful in their careers. In addition to being an excellent language for numerical calculations, Fortran 90 is supported by several libraries such as IMSL, Numerical Recipes, and NAG. It is also a good language for learning general programming principles.

On the other hand, most CHEN 4580 students prefer to use Matlab and the programming language associated with it. The main strength of Matlab is that it is easy to learn and use. Matlab is a good fit for a Numerical Methods course, because it easily performs matrix calculations, a task that arises frequently in CHEN 4580. Furthermore, Matlab is excellent when it comes to plotting and displaying results. It offers numerous styles of graphs and all can be viewed with a simple one-line command. When compared to Fortran 90, Matlab is much better in this respect. Down the road, however, Fortran 90 may be useful to students who face large-scale numerical tasks.

Naturally, the programming modules use both of these languages and progress from using the familiar Fortran 90 to introducing Matlab. To aid in the transition from Fortran 90 to Matlab, we have written a brief tutorial to introduce students to both the front end of Matlab and Matlab m-files. This tutorial is also available on the Computational Methods in Chemical Engineering web site. The two modules used in the sophomore-level courses present the tutorial problems with both Fortran 90 and Matlab programs. For the challenge problem in both of these modules, students are told to use the language with which they are most comfortable. No doubt, most will use Fortran 90.

In contrast, the four junior-level modules emphasize the use of Matlab over Fortran 90. This change encourages students to use Matlab so that it will be a familiar tool by the time they reach CHEN 4580. As the problems become more difficult, the ability to visualize results with graphs becomes more valuable. In addition, these modules require matrix calculations. Because problems in the junior-level modules are more challenging, the easy-to-use Matlab platform will certainly be preferred by students.

The Student Task Force and Module Teaching Assistants

To more easily implement the modules into courses, a student task force was created to write the modules and develop solution keys. Lead by Victor Barocas, this task force ideally has three students each semester, one for each course in which modules are implemented. Although the task force as a whole is responsible for developing the modules, each member of the task force

will also act as a "Module Teaching Assistant" for one course and will focus his/her duties on that course. The Module TA is required to both grade the modules and help students through office hours or help sessions. In addition, this person acts as a liaison between the course instructor and the student task force. We have found that an effective way for the Module TA to help students is to first hold a help session to introduce students to the material presented in the module and answer any initial questions. Then, as the due date approaches, office hours should be held in a computer lab to help students with specific programming tasks. Effectively, the Module TA is a teaching assistant for one homework assignment in the course. Making the Module TA a separate position is preferable, because the instructors and the normal TA's for the courses may not be as familiar with the programming concepts used in the modules.

Developing the modules will require a considerable amount of work from the student task force, but very little work once the modules are written. In our experience, writing and editing a single module, including developing the problems, and making the solution key requires 50-70 hours or 3-5 hours per week over the course of a semester. Considering that three modules must be developed each semester, the student task force will work an estimated twelve hours per week through an entire academic year just in writing the modules. Obviously, developing the modules is the most difficult part of this project. However, the modules may be used multiple times. Once the modules have been written, the student task force is no longer needed. Of course, in using the same module from year to year, the opportunity exists for a sophomore or junior to copy answers from a junior or senior who has already worked the module. In this respect, the modules are no different from many of the assignments in an engineering curriculum. As with any out-of-class assignment, the responsibility of regulating duplication of work lies with the students themselves.

After the modules have been written, only the duties of the Module TA's remain. Between help sessions, office hours, and grading modules, each Module TA will probably only work about fifteen hours or the equivalent of one hour per week over the course of a semester. This estimate is based on an average class size of about 60-70 students for these six classes. Most of the work for Module TA's will be concentrated during the time when the module is used in the class. Again, with three modules each semester, this works out to only three hours of work per week, year-round, in order to implement the programming modules into the curriculum. This is such a small time commitment that it may be possible to borrow TA's from other courses, such as an introductory computing course or a numerical methods course, to also act as Module TA's.

Grading

There are two important points to consider when grading the programming modules. Because the first part of each module contains a great deal of background and no programming, the text in these sections contains fill-in-the-blank questions to guarantee that students are absorbing the material. These questions are relatively easy, but they are numerous. The second point to consider is that the purpose of these modules is more for instruction than evaluation. With this in mind, grading does not seem completely fair. We could never expect to measure the instructional value of a module for each individual. Nevertheless, without a grade students

would have no incentive to work the module. Therefore, some credit should be given just for attempting the programming problems, regardless of whether they were done properly. Any student who has done the programming problems has put forth an effort to learn the material, and the grade should reflect this.

As an example, consider a module that contains twenty fill-in-the-blank questions along with the tutorial and challenge programming problems. A fair grading breakdown might look like this:

- 40 % for the fill-in-the-blank questions (2 pt. each)
- 30 % for completing the programming problems (10 pt. for tutorial problem, 20 pt. for challenge problem)
- 30% for correct answers on the programming problems (10 pt. for tutorial problem, 20 pt. for challenge problem)

A Department-Wide Effort

One problem with implementing programming modules is that they cannot be implemented as an independent project by one or two faculty-members. Because the modules are used in courses throughout the curriculum, they must be a department-wide effort. Naturally, some faculty members will be opposed to implementing programming modules into an engineering curriculum.

Some might argue that programming has little value and will question whether programming should be taught at all. Those opposed to teaching programming argue that professional engineers rarely program because they prefer to use software packages with built-in capabilities. Thus, the time spent teaching programming skills should instead be used to introduce common software packages. Many of these software packages, however, have programming languages associated with them. Excel, Matlab, Mathcad, and LabView all allow their built-in capabilities to be extended through programming. Students and engineers who are unable to program will be limited in their ability to use these software tools.

In addition, programming forces students to learn general problem solving strategies, to organize their thoughts, and to formulate them in the structure of a programming language. Although professional engineers may rarely use their programming skills, they will almost certainly use the problem-solving skills obtained by learning to program. Furthermore, engineers who rarely use programming skills may actually be avoiding opportunities to program, and instead opting for more difficult, less accurate solutions. This programming phobia is unfortunately shared by many engineering students and should by no means be encouraged. Instead, engineering students should have to face their fears of programming by being forced to use the skills repeatedly throughout their curriculum. These programming modules are an excellent way for students to reinforce the skills they learned in introductory computing courses.

Others opposed to introducing programming modules might argue that it will require too much time from both instructor and students and will distract from the material of the course.

Naturally, instructors fear that modules will require too much preparation time. However, as discussed above, the modules require very little effort on the part of the course instructors. The student task force is responsible for developing the modules and solution keys, for grading assignments, and for answering student questions. Time necessary for instructors to edit the modules and help students is minimized by this support.

The instructor's only task would be to make room in the course syllabus for the module. At some point during the course of the semester, preferably toward the end, the instructor must lighten the homework load for students and perhaps allow 10-15 minutes of class time to have a Module TA introduce the module. If the instructors are behind schedule in the course and have not been able to cover all the topics they would have liked to, making room for programming modules may be difficult. However, if the modules are written to include these end-of-the-semester topics, they may be especially valuable to the instructors and students. Furthermore, at the end of the semester, the programming modules may be a good review of the course material. Developing and solving a complex mathematical model requires a deep understanding of the system being modeled. Thus, the modules serve to complement the course material rather than distract from it. If making room in the course syllabus is not an option for some courses, these instructors might consider making the module an optional assignment that would replace a low homework grade. In this way, the instructor would not necessarily need to lighten the students' workload but students would have some incentive for working the module. In addition, this method might allow students to test the modules and give their feedback before implementing the modules as a standard part of the curriculum. By allowing the modules to be used as an extra assignment and removing all responsibility from the instructor, the course instructors are given an offer that is difficult to refuse.

Final Comments

In their first semester, the modules were implemented into courses as optional assignments. Unfortunately, not more than ten students in each course took advantage of the module assignments. But, judging from the students who worked the modules, they were challenging but helpful. The modules were certainly successful in giving students exposure to programming and students who worked the modules will be better prepared for CHEN 4580. As the modules program matures, we expect more complete incorporation into the classes.

Finally, introducing students to computational methods in a curriculum-wide plan such as this is not a task that is restricted to chemical engineering. The programming module concept would be applicable to any engineering curriculum. In fact, implementing computational methods into courses throughout the curriculum may be the best way to teach this topic to engineering students. After all, computational methods are tools to be used with other subjects—would it not be best to teach them in the same way they will be used? With enough background information, these modules could potentially replace a numerical methods course.

Acknowledgments

The authors would like to thank both the University of Colorado Engineering Excellence Fund and well as the Department of Chemical Engineering for their support of this project. We would also like to thank the course instructors for recognizing the value of the modules and implementing them into their courses. Finally, we would like to thank the students who have helped in developing modules and making this project a reality.

Biographical Information

WILLIAM B. PERRY is currently a doctoral student at the Massachusetts Institute of Technology. In 1998, he received his BS in Chemical Engineering from the University of Colorado at Boulder, where he worked as a teaching assistant for Introduction to Engineering Computing and programming module TA. His research interests are in the areas of Biochemical and Biomedical Engineering.

VICTOR H. BAROCAS is an Assistant Professor of Chemical Engineering at the University of Colorado at Boulder. He has taught the undergraduate course CHEN 4580 *Numerical Methods for Process Simulation* for the last two years and has helped promote the broad instruction of numerical methods. His primary research interests are in biomechanics and complex transport systems.

DAVID E. CLOUGH is Professor of Chemical Engineering at the University of Colorado and has been on the faculty there since 1975. From 1986 through 1992 he was Associate Dean of the College of Engineering and Applied Science. His research interests are centered on the optimization and control of chemical processes. He teaches process control, applied statistics, and introductory computing. He lives in a log home in the Rocky Mtns.