10.34 Numerical Methods Applied to Chemical Engineering

MATLAB Tutorial

Kenneth Beers
Department of Chemical Engineering
Massachusetts Institute of Technology
August 1, 2001

The Nature of Scientific Computing

This course focuses on the use of computers to solve problems in chemical engineering. We will learn how to solve the partial differential equations that describe momentum, energy, and mass transfer, integrate the ordinary differential equations that model a chemical reactor, and simulate the dynamics and predict the minimum-energy structures of molecules. These problems are expressed in terms of mathematical operations such as partial differentiation and integration that computers do not understand. All that they know how to do is store numbers at locations in their memory and perform simple operations on them like addition, subtraction, multiplication, division, and exponentiation. Somehow, we need to translate our higher-level mathematical description of these problems into a sequence of these basic operations.

It is logical to develop simulation algorithms that decompose each problem into sets of linear equations of the following form.

$$a_{11}* x_1+ a_{12}*x_2 + \ldots + a_{1n}*x_n = b_1$$

$$a_{21}*x_1 + a_{22}*x_2 + \ldots + a_{2n}*x_n = b_2$$

$$.$$

$$.$$

$$.$$

$$a_{n1}*x_1 + a_{n2}*x_2 + \ldots + a_{nn}*x_n = b_n$$

A computer understands how to do the operations found in this system (multiplication and addition), and we can represent this set of equations very generally by the matrix equation Ax = b, where $A=\{a_{ij}\}$ is the matrix of coefficients on the left hand side, x is the solution vector, and b is the vector of the coefficients on the right hand side. This general representation allows us to pass along, in a consistent language, our system-specific linear equation sets to pre written algorithms that have been optimized to solve them very efficiently. This saves us the effort of coding a linear solver every time we write a new program. This method of relegating repetitive tasks to re-usable, pre written subroutines makes the idea of using a computer to solve complex technical problems feasible. It also allows us to take advantage of the decades of applied mathematics research that have gone into developing efficient numerical algorithms. Scientific programs typically involve problem-specific sections that perform the parameter input and results output, phrase the problem into a series of linear algebraic systems, and then the program spends most of its execution time solving these linear systems. This course focuses primarily on understanding the theory and concepts fundamental to scientific computing, but we also need to know how to translate these concepts into working programs and to combine our problem-specific code with pre written routines that efficiently perform the desired numerical operations.
So, how do we instruct the computer to solve our specific problem? At a basic level, all a computer does is follow instructions that tell it to retrieve numbers from specified memory

locations, perform some simple algebraic operations on them, and store them in some (possibly new) places in memory. Rather than force computer users to deal with details like memory addresses or the passing of data from memory to the CPU, computer scientists develop for each type of computer a program called a *compiler* that translates ãhuman-levelä code into the set of detailed machine-level instructions (contained in an *executable* file) that the computer will perform to accomplish the task. Using a compiler, it is easy to write code that tells a computer to do the following :

1. Find a space in memory to store a real number x
2. Find a space in memory to store a real number y
3. Find a space in memory to store a real number z
4. Set the value of x to 2
5. Set the value of y to 4
6. Set the value stored at the location z to equal 2*x + 3*y, where the symbol * denotes multiplication

In FORTRAN, the first modern scientific programming language that, in modified form - commonly FORTRAN 77, is still in wide use today, you can accomplish these tasks by writing the code :

```
REAL x, y, z
x = 2
y = 4
z = 2*x + 3*y
```

By itself, however, this code performs the desired task, but does not provide any means for the user to view the results. A full FORTRAN program to perform the task and write the result to the screen is :

```
IMPLICIT NONE
REAL x, y, z
x = 2
y = 4
z = 2*x + 3*y
PRINT *, 'z = ',z
END
```

When this code is compiled with a FORTRAN 77 compiler, the output to the screen from running the *executable* is : z = 16.0000. Compiled programming languages allow only the simple output of text, numbers, and binary data, so any graphing of results must be performed by a separate program. In practice, this requirement of writing the code, storing the output in a file with the appropriate format, and reading this file into a separate graphing or analysis program leads one to use for small projects "canned" software such as EXCEL that are ill-suited for technical computing; after all, EXCEL is intended for business spreadsheets!

Other compiled programming languages exist, most being more powerful than FORTRAN 77, a legacy of the past that is retained mostly due to the existence of highly efficient numerical routines written in the language. While FORTRAN 77 lacks the functionality of more modern languages, in terms of execution speed it usually has the advantage. In the 80's and 90's, C and C++ became highly popular within the broader computer science community because they allow one to organize and structure data more conveniently and to write highly-modular code for large programs. C and C++ have never gained the same level of popularity within the scientific computing community, mainly because their implementation has been focused more towards robustness and generality with less regard for execution speed. Many scientific programs have comparatively simple structures so that execution speed is the primary concern. This situation is changing somewhat today; however, the introduction of FORTRAN 90 and its update FORTRAN 95 have given the FORTRAN language a new lease on life.

FORTRAN 90/95 includes many of the data structuring capabilities of C/C++, but was written with a technical audience in mind. It is the language of choice for parallel scientific computing, in which tasks are parceled during execution to one or more CPU's. With the growing popularity of dual processor workstations and BEOWOLF-type clusters, FORTRAN 90/95 and variants such as High Performance Fortran remain my personal *compiled* language of choice for heavy-duty scientific computing.

Then why does this course use MATLAB instead of FORTRAN 90? FORTRAN 90 is my choice among *compiled* languages; however, for ease of use, MATLAB, an *interpreted* language, is better for small to medium jobs. In *compiled* languages, the "human-level" commands are converted directly to machine instructions that are stored in an executable file. Run-time execution of the commands does not take place until all of the compilation process has been completed (run-time debugging not excepted). In a *compiled* language, one needs to learn the commands for the input/output of data (from the keyboard, to the screen, to/from files) and for naming variables and allocating space for them in memory (like the command real in FORTRAN). *Compiled* languages are developed with the principle that the language should have a minimum amount of commands and syntax, so that any task that may be accomplished by a sequence of more basic instructions is not incorporated into the language definition but is rather left to a subroutine. Subroutine libraries have been written by the applied mathematics community to perform common numerical operations (e.g. BLAS and LAPACK), but to access them you need to link your code to them through operating system-specific commands. While not conceptually difficult, the overhead is not insignificant for small projects.

In an *interpreted* language, the developers of the language have already written and compiled a master program, in our case the program MATLAB, that will interpret our commands to the computer ãon-the-flyä. When we run MATLAB, we are offered a window in which we can type commands to perform mathematical calculations. This code is then interpreted line-by-line (by machine-level instructions) into other machine-level instructions that actually carry out the computations that we have requested. Because MATLAB has to interpret each command one-by-one, we will require more machine-level instructions to perform a certain job that we would with a *compiled* language. For demanding numerical simulations, where we need to use the resources of a computer as efficiently as possible, *compiled* languages are therefore superior.

Using an *interpreted* language has the benefit; however, that we do not need to compile the code before-hand. We can therefore type in our commands one-by-one and watch them be performed (this is very helpful for finding errors). We do not need to link our code to subroutine libraries, since MATLAB, being pre compiled, has all the machine-level instructions it needs readily at-hand. FORTRAN 77/90/95, C, and C++ cannot make graphs, so if we want to plot the results from our program, we need to write data to an output file that we use as input to yet another graphics program. By contrast, the MATLAB programmers have already provided graphics routines and compiled them along with the MATLAB code interpreter, so we do not need this additional data transfer step. An *interpreted* language can provide efficient and complex memory management utilities that, by operating behind a curtain, shield the programmer from having to learn their complicated syntax of usage. New variables can therefore be created with dynamic memory allocation without requiring the user to understand pointers (variables that point to memory locations), as is required in most *compiled* languages. Finally, since MATLAB was not developed with the principle of minimum command syntax, it contains a rich collection of integrated numerical operations. Some of these routines are designed to solve linear problems very efficiently. Others operate at a higher level, for example taking as input a function f(x) and returning the point x_0 that has $f(x_0)=0$, or integrating the ordinary differential equation $dx/dt = f(x)$ starting from a value of x at $t=0$.

For these reasons, one can code more efficiently in *interpreted* languages than in *compiled* languages (McConnell, Steve, *Code Complete*, Microsoft Press, 1993 and Jones, Capers, *Programming Productivity*, McGraw-Hill, 1986), at the cost of slower execution due to the extra interpreting step for each command. But, we have noted before that execution speed is an important consideration in scientific computing, so is this acceptable? MATLAB has several features to alleviate this situation. Whenever MATLAB first runs a subroutine, it saves the results of the interpreting process so that successive calls do not have to repeat this work. Additionally, one can reduce the interpretation overhead by minimizing the number of command lines, a practice which incidentally leads to good programming style for FORTRAN 90/95. As an example, let us take the operation of multiplying a M by N matrix A with an N by P matrix B to form a M by P matrix C. In FORTRAN 77 we would first have to declare and allocate memory to store the A, B, and C matrices (as well as the counter integers i_row, i_col, and i_mid), and then, perhaps in a subroutine, execute the code :

```
DO i_row = 1, M
  DO i_col = 1, N
    C(i_row,i_col) = 0.0
    DO i_mid = 1, P
      C(i_row,i_col) = C(i_row,i_col) + A(i_row,i_mid)*B(i_mid,i_col)
    ENDDO
  ENDDO
ENDDO
```

If we simply translated each line, one-by-one, from FORTRAN 77 to MATLAB, we would have the code segment :

```
for i_row = 1:M
  for i_col = 1:N
    C(i_row,i_col) = 0;
    for i_mid = 1:P
      C(i_row,i_col) = C(i_row,i_col) + A(i_row,i_mid)*B(i_mid,i_col);
    end
  end
end
```

This code performs the task in exactly the same manner as FORTRAN 77, but now each line must be interpreted one-by-one, adding a considerable overhead. It would seem that we would be better off with FORTRAN 77; however, in MATLAB the language is extended to allow matrix operations so that we could accomplish the same task with the single command : C = A*B. We would not even have to pre allocate memory to store C, this would be automatically handled by MATLAB. The MATLAB approach is greatly to be preferred, and not only because it accomplishes the same task with less typing (and chance for error!). The FORTRAN 77 code, relying on basic scalar addition and multiplication operations, is not very easy to parallelize. It instructs the computer to perform the matrix multiplication with an exact order of events that the computer is constrained to follow. The single command C = A*B requests the same task, but leaves the computer free to decide how to accomplish it in the most efficient manner, for example, by splitting the problem across multiple processors. One of the main advantages of FORTRAN 90/95 over FORTRAN 77 is that it also allows these whole array operations (the corresponding FORTRAN 90/95 code is C = MATMUL(A,B)), so that writing fast MATLAB code rewards the same programming style as does FORTRAN 90/95 for producing code that is easy to parallelize.

MATLAB also comes with an optional compiler that converts MATLAB code to C or C++ and that can compile this code to produce a stand-alone *executable*. We therefore can enjoy the ease of programming in an *interpreted* language, and then once the program development is complete, we can take advantage of the efficient execution and portability offered by *compiled* languages. Alternatively, given the tools of the compiler, we can combine MATLAB code and numerical routines with FORTRAN or C/C++ code. Given these advantages, MATLAB seems a strong choice of language for an introductory course in scientific computing.

MATLAB Tutorial Table of Contents

This tutorial is presented with a separate webpage for each chapter. The commands listed in the tutorial are explained with comment lines starting with the percentage sign %. These commands may either be typed or pasted one-by-one into an interactive MATLAB window. Further information about a specific command can be obtained by typing help followed by the name of the command. Typing helpwin brings up a general help utility, and helpdesk provides links to extensive on-line documentation. For further details, consult the texts found in the Recommended Reading section of the 10.34 homepage.