Editor: Norman Chonacky, CiSE-Editor@aip.org

3Ms for Instruction, Part 2:

Maple, Mathematica, and Matlab

By Norman Chonacky and David Winch

OUR INTENT WITH THIS TECHNOLOGY REVIEW IS TO PRESENT A FRAMEWORK THAT HELPS EDUCATORS MAKE THEIR OWN CRITICAL COMPARISON OF MAPLE, MATHEMATICA, AND MATLAB AS CANDIDATE COMPUTATIONAL PRODUCTIVITY tools for use in their instructional programs. This is an alternative to our providing a critical comparison of our own, as is conventional in a review. In the first installment, we provided a common set of talking points—concrete, understandable, existing applications as well as an idealized "paradigmatic" example—around which to build this framework. We also defined a particular subset of issues that undergraduate science and engineering educators face regarding computational technology. In this issue, we conclude this framework-building strategy by defining a compact, common feature set in which we can finally describe in some comparable detail how Maple, Mathematica, and Matlab work.

To illustrate these features and to refer them to science and engineering contexts, we chose numerical, rather than symbolic, computational examples, each showing the packages performing identical tasks. One cost of this choice is that we can't discuss all the wonderful symbolic computation capabilities of Mathematica and Maple. (Matlab's symbolic computational tools are a subset of Maple.) Although we can justify this choice based on the testimony of both neophyte and veteran users from the educational community whom we interviewed, it highlights a major compromise, one of many, needed to create this type of technology review. As an experiment, we set out to give a broad scope of readers the material they'll need to address educational issues, along with helpful and concise evaluation guidance. We hope we've struck a proper balance, avoiding both superficiality and technicality.

Development and Delivery Environments

What's it like to work with these packages? Users who wish to create or modify content must work within the associated development environments. Such users will be both faculty developing educational materials and students writing computational code, the only exception being students using applications mediated by custom-created, application-specific graphical user interfaces (GUIs).

Both Maple and Mathematica supply standard interfaces for their development environments that are already GUIs of a kind. These consist of book-like content windows that hold interactive text and graphics; these content windows also have pull-down menus from a menu bar and palettes of tools. In Mathematica, these palettes are movable and can be custom-created as part of the development environment or attached to applications. Figures 1 and 2 are screenshots of the Mathematica and Maple development interfaces, respectively.

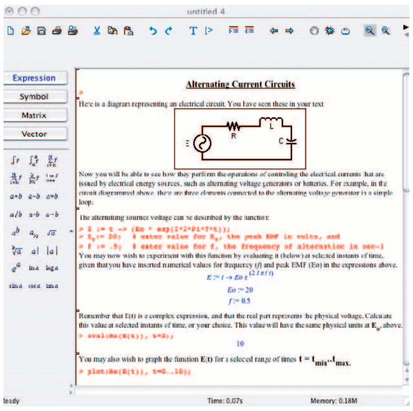
Matlab's development environment is quite different. Basically, it has a command line displayed in one of several windows. The main, circumscribing window (called the Desktop) has pull-down menus from an overhead menu bar. The default Desktop configuration, shown in Figure 3, is subdivided into several partitions, each of which is itself a resizable window. The partitions contain a command line, a command stack, and a directory tree.

Developers using Mathematica or Maple enter the computing objects—such as variables, operations, descriptive text, and so on—into segmented cells. In Maple, these have a single logical level, whereas they can be nested hierarchically in Mathematica. These cells extend the command-line concept by encapsulating commands, but they also integrate narrative text, making their aggregate—the Mathematica Notebook or the Maple Worksheet—similar to interactive books. Any entities the developer creates in a session, such as variable names or session histories, are maintained implicitly by the system; however, commands, sometimes several commands, are needed to explicate them.

In contrast, developers working in the Matlab environment use a conventional command line. They encapsulate command sets by placing them in separate files, which is one reason to have a file directory partition visible. In general, narrative text can't be promiscuously mixed with commands and must be restricted to comments. The system segregates computing objects from session and environmental information, displaying them in other Desktop windows; thus, the lists of entities the developer creates in a session are both explicitly maintained and continually viewable without the need for commands.

Both of these interface design motifs—books and Desktop—let the developer interactively conduct calculations using a set of commands and variables with a distinctive vocabulary and syntax (essentially, a programming language). But each motif offers a different deployment strategy for exporting programmed calculations. Mathematica and Maple produce books of intermixed narrative and code that a student can be given to use with or without the development tools, thus the use experiences of developer and user can be quite different. Matlab produces code files that can be stored in one of the host's directories. That directory must be inserted into Matlab's list of search paths and then executed from the command line in its Desktop window by typing in its file name. Thus the use of Matlab code looks and works essentially the same for both user and developer, requiring the user to have at least some measure of expertise with the development environment.

Figures 4 and 5 amply illustrate the differences in running applications that are program/narrative combinations between the two motifs. Mathematica's Notebook or Maple's Worksheet can look "cleaned" of development artifacts, such as tools (see Figure 4). Users navigate through these books by reading the narrative, changing parameters for calculation methods by editing the text, and executing computations either by moving to a single command or selecting "cells" of code.



Alternating Current Circuits

Here is a diagram representing an electrical circuit. You have seen these in your text

Now you will be able to see how they perform the operations of controling the electrical currents that are issued by electrical energy sources, such as alternating voltage generators or batteries. For example, in the circuit diagrammed above, there are three elements connected to the alternating voltage generator in a simple loop.

The alternating sourse voltage can be described by the function:

```
> E := t -> (Eo * exp(I*2*Pi*f*t));
> E0:= 20;   # enter value for E0, the peak EMF in volts, and
> f := .5;   # enter value for f, the frequency of alternation in sec-1
```

You may now wish to experiment with this function by evaluating it (below) at selected instants of time, given that you have inserted numerical values for frequency (*f*) and peak EMF (Eo) in the expressions above.

$$E := t \to Eo\, e^{(2 I \pi f t)}$$

$$Eo := 20$$

$$f := 0.5$$

Remember that E(t) is a complex expression, and that the real part represents the physical voltage. Calculate this value at selected instants of time, or your choice. This value will have the same physical units at E_0, above.

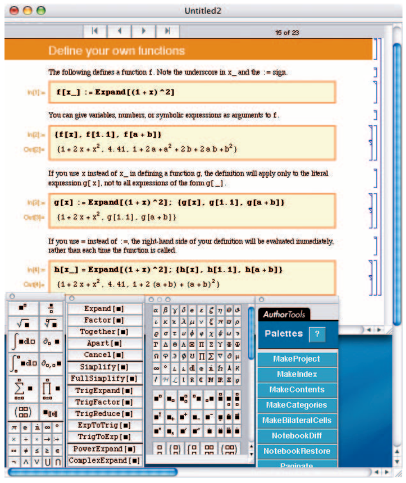
```
> evalf(Re(E(t)), t=2);
```

10

You may also wish to graph the function E(t) for a selected range of times $t = t_{min}..t_{max}$.

```
> plot(Re(E(t)), t=0..10);
```

Figure 1. Maple development environment. The palette displayed in the user interface is one of four, here showing Expressions.



The following defines a function f. Note the underscore in x_ and the := sign.

```
In[1]:= f[x_] := Expand[(1 + x)^2]
```

You can give variables, numbers, or symbolic expressions as arguments to f.

```
In[2]:= {f[x], f[1.1], f[a + b]}
Out[2]= {1 + 2 x + x^2, 4.41, 1 + 2 a + a^2 + 2 b + 2 a b + b^2}
```

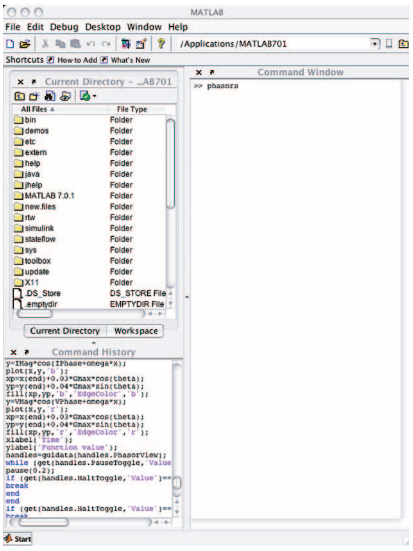
If you use x instead of x_ in defining a function g, the definition will apply only to the literal expression g[x], not to all expressions of the form g[_].

```
In[3]:= g[x] := Expand[(1 + x)^2]; {g[x], g[1.1], g[a + b]}
Out[3]= {1 + 2 x + x^2, g[1.1], g[a + b]}
```

If you use = instead of :=, the right-hand side of your definition will be evaluated immediately, rather than each time the function is called.

```
In[4]:= h[x_] = Expand[(1 + x)^2]; {h[x], h[1.1], h[a + b]}
Out[4]= {1 + 2 x + x^2, 4.41, 1 + 2 (a + b) + (a + b)^2}
```

Figure 2. Mathematica development environment. Here we see three tool palettes and a master palette that controls the others (it also includes a collection of "wizards" listed as author tools).

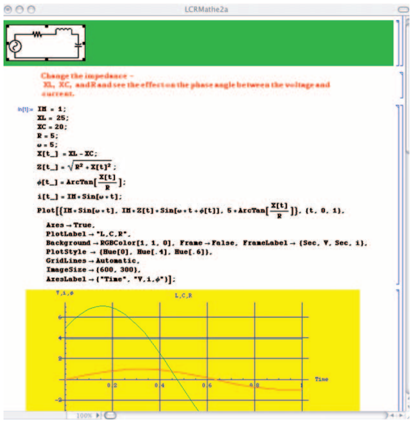

MATLAB
File Edit Debug Desktop Window Help
/Applications/MATLAB701
Shortcuts How to Add What's New
Current Directory - ...AB701
Command Window
Current Directory
Workspace
Command History
Start

All Files	File Type
bin	Folder
demos	Folder
etc	Folder
extern	Folder
help	Folder
java	Folder
jhelp	Folder
MATLAB 7.0.1	Folder
new.files	Folder
rtw	Folder
simulink	Folder
stateflow	Folder
sys	Folder
toolbox	Folder
update	Folder
X11	Folder
.DS_Store	DS_STORE File
.emptydir	EMPTYDIR File

```
>> phasors
```

```
y=IMag*cos(IPhase+omega*x);
plot(x,y,'b');
xp=x(end)+0.03*Gmax*cos(theta);
yp=y(end)+0.04*Gmax*sin(theta);
fill(xp,yp,'b','EdgeColor','b');
y=VMag*cos(VPhase+omega*x);
plot(x,y,'r');
xp=x(end)+0.03*Gmax*cos(theta);
yp=y(end)+0.04*Gmax*sin(theta);
fill(xp,yp,'r','EdgeColor','r');
xlabel('Time');
ylabel('Function value');
handles=guidata(handles.PhasorView);
while (get(handles.PauseToggle,'Value
pause(0.2);
if (get(handles.HaltToggle,'Value')==
break
end
end
if (get(handles.HaltToggle,'Value')==
break
```

Figure 3. Matlab development environment. The default configuration shows the package's content window, called the Desktop, containing command line, history, and directory partitions.



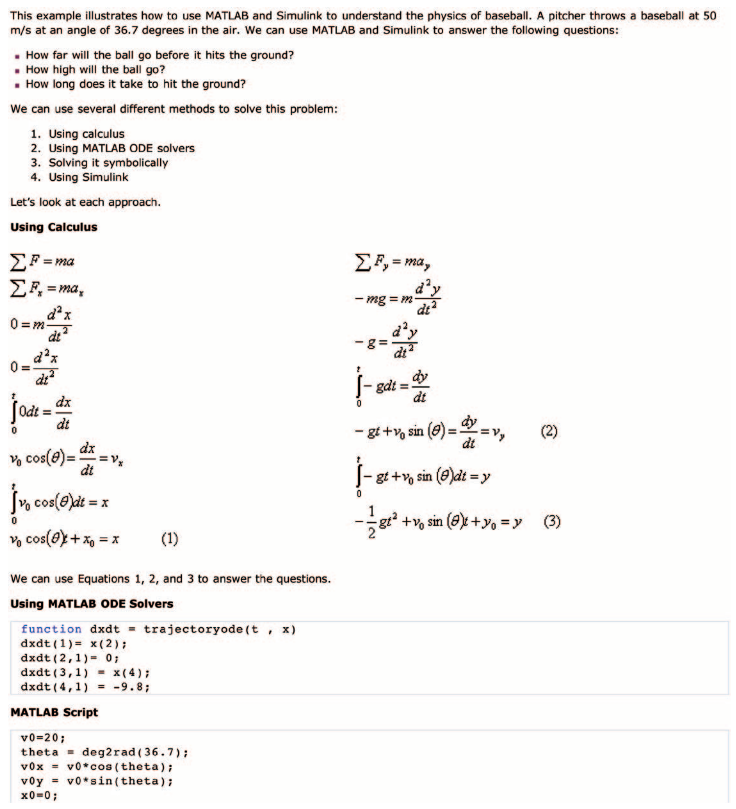
Change the impedance –
XL, XC, and R and see the effect on the phase angle between the voltage and current.

```
In[1]:= IM = 1;
        XL = 25;
        XC = 20;
        R = 5;
        ω = 5;
        X[t_] = XL - XC;
        Z[t_] = √(R² + X[t]²);
        ϕ[t_] = ArcTan[X[t]/R];
        i[t_] = IM * Sin[ω * t];
        Plot[{IM * Sin[ω * t], IM * Z[t] * Sin[ω * t + ϕ[t]], 5 * ArcTan[X[t]/R]}, {t, 0, 1},
          Axes → True,
          PlotLabel → "L,C,R",
          Background → RGBColor[1, 1, 0], Frame → False, FrameLabel → {Sec, V, Sec, i},
          PlotStyle → {Hue[0], Hue[.4], Hue[.6]},
          GridLines → Automatic,
          ImageSize → {600, 300},
          AxesLabel → {"Time", "V,i,ϕ"}];
```

Figure 4. Mathematica Notebook. The narrative guides the user on how to interact with the Notebook. In this case, selecting one or more lines of code and pressing the Enter key would execute those commands.

Matlab's program code and expository narrative can be placed in an exogenous text file (see Figure 5). Being outside of Matlab, its code portions aren't executable. The code's point of entry into the computational engine is the command line of a command window, thus in this scenario, the text form of the programmed commands (for example, "Using MATLAB ODE Solvers" in the middle of Figure 5) would need to be cut and pasted into a command line to perform the indicated calculation.

Recently, the three companies that produce these packages have made concerted efforts to provide developers with GUIs. In contrast to the book/Desktop distinction, the GUIs offer similar appearances and function-



This example illustrates how to use MATLAB and Simulink to understand the physics of baseball. A pitcher throws a baseball at 50 m/s at an angle of 36.7 degrees in the air. We can use MATLAB and Simulink to answer the following questions:

- How far will the ball go before it hits the ground?
- How high will the ball go?
- How long does it take to hit the ground?

We can use several different methods to solve this problem:

1. Using calculus
2. Using MATLAB ODE solvers
3. Solving it symbolically
4. Using Simulink

Let's look at each approach.

Using Calculus

$$\sum F = ma$$
$$\sum F_x = ma_x$$
$$0 = m\frac{d^2x}{dt^2}$$
$$0 = \frac{d^2x}{dt^2}$$
$$\int_0^t 0dt = \frac{dx}{dt}$$
$$v_0 \cos(\theta) = \frac{dx}{dt} = v_x$$
$$\int_0^t v_0 \cos(\theta) dt = x$$
$$v_0 \cos(\theta)t + x_0 = x \qquad (1)$$

$$\sum F_y = ma_y$$
$$-mg = m\frac{d^2y}{dt^2}$$
$$-g = \frac{d^2y}{dt^2}$$
$$\int_0^t -gdt = \frac{dy}{dt}$$
$$-gt + v_0 \sin(\theta) = \frac{dy}{dt} = v_y \qquad (2)$$
$$\int_0^t -gt + v_0 \sin(\theta) dt = y$$
$$-\frac{1}{2}gt^2 + v_0 \sin(\theta)t + y_0 = y \qquad (3)$$

We can use Equations 1, 2, and 3 to answer the questions.

Using MATLAB ODE Solvers

```
function dxdt = trajectoryode(t , x)
dxdt(1)= x(2);
dxdt(2,1)= 0;
dxdt(3,1) = x(4);
dxdt(4,1) = -9.8;
```

MATLAB Script

```
v0=20;
theta = deg2rad(36.7);
v0x = v0*cos(theta);
v0y = v0*sin(theta);
x0=0;
```

Figure 5. Matlab's "textbook." The text provides users with explanatory narrative and M-code to cut and paste into the command line of a command window. (Figure courtesy of Student Center Homework, "Physics of Baseball," www.mathworks.com/academia/student_center/homework/simulink/sim_ex4.html.)

ality across all three products. They can contain textboxes, buttons, and mouse-controlled devices such as sliders for user input, and they all provide interactive graphics for output. We've provided examples of GUIs produced by each package in the first installment of this review. Although GUIs delimit the learning experience by "hiding" the underlying code, they facilitate it by simplifying data input and control functions, allowing users to explore the computation's parameter space. In the rush to expand GUI presence, Matlab alone (as of this writing) has a specialized development environment that generates the actual code for producing GUIs; the other two packages offer conventional application programmer interfaces (APIs) for this purpose.

Common Feature Sets

By considering our previous examples across all three application categories and the results of our interviews with instructors regarding their use experiences, we were able to extract a feature set of the most significant objects and operations common to all three packages. This feature set consists of programming language, program code management and debugging, formatting, and interactive graphical input and output.

Programming Language

Although each package's programming language embodies a complete set of data and control primitives, details among the three differ. Consequently, any given type of computation might be more concise in one package than in another. This presents two conundrums in devising examples for the common feature set. Should we choose optimal codes for each package or codes most faithfully parallel to one another? Should we recruit "expert" coders to construct the examples or do them ourselves, based on several weeks of our preparatory study? Our decision: do them ourselves as parallel examples. Expert codes and tailored examples will await the next review in this series, "3Ms for Research and Development." Our examples here instead provide a realistic, unpolished account of what experienced programmers can produce in a first encounter with these packages.

Figure 6 shows how to construct a matrix, alternatively by enumeration or computation. The boxes following the statements contain snapshots of what the resulting output of either method looks like.

It's worth mentioning that the treatment afforded a *matrix* differs in each package—in Maple, it's a matrix; in Mathematica, it's a list; and in Matlab, it's a number. These treatments also reflect each package's underlying character. Maple, for example, uses different type names to distinguish scalars, vectors, and matrices. It also uses conventional mathematical notation in its output, revealing the value it places on mathematical convention in the naming and visual appearance of objects and processes. Matlab has a more pragmatic sense of what's useful by favoring computational conciseness. It treats every datum as a matrix, for example, but of different orders for scalars, vectors, and higher-order tensors. Mathematica reveals something of its underlying sophisticated computation-theoretic values by favoring abstract structures—it treats a matrix as a list of embedded lists, for example.

Figure 7 shows how to construct a function from a simple expression. Maple uses conventional mathematical symbols in its command vocabulary as well as in the output display in which a function is portrayed as a mapping and the exponential function as "e" raised

In Maple:

```
    A:= < <2 | 4> , <6 | 8> >                         # Matrix construction and initial valuation
 or
    n :=2 : f:= ( i, j )  -> 2*( j + 2*(i-1) ) :      # initial valuation function
    A:= Matrix( n, f );                               # Matrix construction
```

$$\boxed{A := \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}}$$

In Mathematica:

```
    A = {  {2 , 4}, {6 , 8}  }                        (* List construction and initial valuation*)
 or
    A = Table [ 2i + 4(j - 1), {j, 2}, {i, 2}]        (* List construction and initial valuation*)
```

```
Out[2]  =  {{2,  4},  {6,  8}}
```

In Matlab:

```
    A = [ [ 2  4 ] ; [ 6  8 ] ]                       % Number construction and initial valuation
 or
    nn = 2;  Atrsp = zeros( nn, nn )                  % Number construction and initial valuation
     for kk = 1 : (nn* nn) ; Atrsp (kk) = 2*kk;       % Number valuation
     end
    A = Atrsp'                                        % Number transpose
```

```
A =
      2   4
      6   8
```

Figure 6. Matrix variable. Using each package, we show language statements and (boxed) results for the programmed construction of a simple 2 × 2 matrix.

to a power. Mathematica also uses the exponential function's conventional appearance in its output, but has an idiosyncratic syntax for function construction. Matlab expresses something of its underlying computational strategy in its use of the @ symbol, suggesting "indirect addressing" of locations in computer memory. In fact, its function constructor equates to a function "handle" (Matlab vocabulary) to be used in referencing syntax. Matlab doesn't use conventional mathematical notation for exponential functions in its output.

Maple and Mathematica use color and labels to distinguish input from output. Additionally, Mathematica indexes its input and output from when the session starts, allowing reference to previous entries and results in that session. Matlab makes no such visual or indicial distinctions.

At this point, you can already see some obvious syntactical differences in meaning for symbols and delimiters among the packages, but their implications for educational development and uses can be quite subtle. Mathematica doesn't use the symbol "E" for the complex function's name; instead, "E" is a restricted symbol used only for the base of the natural logarithms. Similarly, Maple and Mathematica reserve "I" and Matlab saves "i" (and "j") for designating imaginary numbers ($\sqrt{-1}$). Restricted symbols exist in all programming languages and are useful in a variety of contexts, but they can be a nuisance to programmers and a source of confusion to novice users of interactive tutorials. Maple acknowledges this by no longer restricting "E," using instead the exponential function exp(1) as the base of the natural logarithms.

For episodic users, these differences make promiscuous movement from one package to another particularly frustrating and even painful. These subtle variances also explain why, as we found in our interviews, individuals typically use only one package even when all three are available, and why many respondents viewed the issue of departmental or institutional adoption of a single package as significant.

The interactive interfaces for all three packages share a common trait: users create new variables by simply referring to them for the first time with any unreserved name. The symbolic-oriented packages (Mathematica and Maple) create these new variables as symbols, not values. Assigning values to these symbols changes them from symbolic to numerical; in all three packages, assigning a value to an existing variable name resets it to

In Maple:

```
E := t -> (Eo * exp( I * 2 * Pi * f * t ) );     # function definition
Eo := 20;                                        # assign value to variable Eo
f  := .5;                                        # assign value to variable f
```

$$\boxed{\begin{array}{l} E := t \to Eo\ e^{(2 I \pi f t)} \\ Eo := 20 \\ f := 0.5 \end{array}}$$

In Mathematica:

```
Eo = 20;                            (* assign value to variable Eo *)
f  = .5;                            (* assign value to variable f *)
Einst[ t_ ] := Eo * Exp[ 2 I Pi f t ];   (* function definition *)
Einst[t]
```

```
Out[1]  = 20 e^{3.14159 i t}
```

In Matlab:

```
Eo = 20;                              % assign value to variable Eo
f  = .5;                              % assign value to variable f
E = @ ( t )   Eo * exp( i * 2 * Pi * f * t);   % function definition
```

```
E =
      @ ( t )   Eo * exp(i * 2 * pi * f * t)
```

Figure 7. A function from a simple expression. Using each package, we show language statements and (boxed) results for the construction of a (complex) function.

the new value. Unfortunately, this feature can lead users to inadvertently change variable values and obtain unexpected results. Encapsulation mechanisms (such as *procedures*, *objects*, and so on) are a common way to limit the scope of variables and thus the conflicts that can arise from using competing variable names. They're also useful for grouping and associating command sequences. Figure 8 illustrates how "procedural" methods work in each of the three packages.

Encapsulation in Mathematica can be achieved by using the package's Module construct. The brackets [ ] on the right-hand side delimit the modular code, which is assigned to a function, here named either `seriz` or `parallelz`; the braces { } delimit the local variables *x* and *y* and assign them (initial) values. Note that Output[1] and Output[2], returned after entering the Module definition, express symbolic objects—z1 and z2—because they haven't been evaluated yet. Once these formally symbolic objects are evaluated, as in Input[3], the results are the values of `seriz` and `parallelz` expressed in Output[4] and Output[5].

Strictly speaking, Mathematica doesn't have procedures (or, at least, doesn't refer to them as such). This is consistent with its goal, stated openly in the introductory chapter of its manual, of changing the way users think about mathematics and programming. This attitude can be an instructional asset or liability, depending on whether you view it as abetting or subverting your educational goals.

Matlab also eschews procedures, using a function construct to achieve encapsulation instead. The normal way to create a Matlab function is to place the code in a separate file, a process that we've already detailed in the section earlier on development environments. In our example in Figure 8, we can use an alternative method because each function's tasks can be encoded in a single expression. This then allows the use of an "anonymous" (Matlab vocabulary) function definition, a shortcut that still treats its variables *x* and *y* as local.

Note that the output following the anonymous function definitions, the first entry in the Matlab scenario in Figure 8, treats the two functions as belonging to a group with a common surname, `combo`. Matlab makes this association between the functions because of the form of our choice for their names: specifically, `surname.name`. Matlab treats the second parts (`name1`, `name2`, and so on) as named fields of a common record, `surname`. This adopts the syntactical convention of "record" similar to what programming languages do and italicizes a persistent personality trait of Matlab—its close relationship to conventional programming.

Maple uses an encapsulation concept similar to Mathematica in its Module construct, which Maple defines as a generalization of the procedure concept. Whereas the procedure associates "a sequence of commands with a single command," the module associates "related functions and data." In Figure 8, the module definition first names local variables in a declaration, then the export declaration, followed by the Maple statements that perform calculations and assign values to the export names. Export names establish local variables, but they're also available to the module's clients once that module is instantiated. In effect, this allows retrieval of the export values, subsequent to the Module's instantiation, by references to the export variables that use the "member selection" operator (`:-`), as indicated in the example. The overall effect is similar to using the naming convention in Matlab, except that the construct is more elaborate. Maple Modules, for instance, not only achieve encapsulation and packaging, but also permit object modeling and generic programming.

Programming languages are most important for computational instruction applications, because both developers and students must use languages extensively. They're also important for developers of both simulation and tutorial materials, more so in the former and less in the latter, because they're more computationally intensive. For students, they're more or less important in simulations and tutorials, depending on how much direct access to code the developer chooses to give to the user.

Program Code Management and Debugging

All three packages provide the means to manage and debug code, but they differ somewhat in their design—consistent with the packages' previously discussed individual personalities.

Each provides an interactive interface as its primary method for communicating with its respective computation engine, or kernel. These kernels let users test commands and simple routines on the fly by providing error messages linked to help messages. To varying degrees, all three packages support "extensible help" by letting developers create customized documentation or help information. This is especially useful in educational settings, serving both the episodic developer of instructional materials and the student learning to use a system for his or her own computations.

The packages all have conventional debuggers that are useful for engineering more complex code segments. These utilities encapsulate the code-execution process, permitting the user to set breakpoints, step through routines, view the states of variables as they change during execution, and so on. Additionally, the packages have facilities for creating libraries and archives of (debugged) code files.

In Mathematica:

```
In[1]:=   seriz[z1_,z2_] := Module[{x = z1, y = z2}, (x+y)]
          parallelz[z1_, z2_] :=
           Module[{x = z1, y = z2}, (xy) / (x+y)]
Out[1] = z1 + z2
Out[2] = z1z2/(z1+z2)
In[3]:=   R =10; Xc =(1/{2 * I * Pi * 0.02))
          seriz[R, Xc]
          parallelz[R, Xc]
Out[4] = 10 – 7.95775 i
Out[5] = 3.87727 – 4.87232 i
```

In Matlab:

```
>> combo.series = @(x, y) x + y ; combo.parallel=
       @ (x, y) x * y/(x + y)
combo =
      series: @ (x, y) x + y
    parallel: @ (x, y) x *y/(x + y)
>> R=10; Xc= 1/(2*i*pi*0.02);
>> combo.series (R, Xc)
ans =
  10.0000 - 7.9577i
>> combo.parallel (R,Xc)
ans =
  3.8773 - 4.87231
```

In Maple:

```
combination := module()
      local x, y;
      export series, parallel;
      series := (x, y) -> simplify(x+y);
      parallel := (x, y) -> simplify((x*y)/( x+y));
end module;

R:= 10; Xc:= (-1/(2*I*Pi*0.02));
combination:-series (R, Xc);
combination:-parallel (R, Xc);
```

$$R := 10$$

$$Xc := \frac{-25.00000000\, I}{\pi}$$

combination:=module()export series, parallel; end module

10.00000000 – 7.957747150 *I*

3.877266367 – 4.872316614 *I*

Figure 8. Procedural definitions. Using each package, we show language statements and (boxed) results for the equivalent computations of the resulting impedances in the cases of parallel and series connections of both a resistor and a capacitor.

These operations work more or less the same across all three packages and are similar to those provided in other code development environments.

In an effort aimed at interoperability, all the packages provide support for integrating external code into their applications and vice versa (support for external applications to call the package code). These capabilities are important for any developer intent on using other software as part of the mix.

Maple. The Maple Worksheet—Maple's standard interactive interface—is a generalized type of GUI that allows the entry and execution of commands and simple code clusters within cells in the Worksheet window. Specialized Maple GUIs created as code objects are called Maplets; the Maplet development environment is implemented as a set of specialized commands in a separate subpackage that is integrated into the standard Maple code development environment. Maple supports interoperability for C, Java, and Visual Basic. OpenMaple is an API into Maple that lets users write programs in C (and thus C++), Java, and VisualBasic (VB) and then call Maple routines from within those programs. Maple also has built-in functionality to automatically generate optimized code in C, Fortran, Java, Visual Basic, and Matlab (M-code), given a Maple equation or a Maple program.

Mathematica. The Mathematica Notebook—Mathematica's standard interactive interface—is also a type of GUI. Predictably, Mathematica's debugging facilities are elaborate and exotic. The package doesn't have a debugger per se; rather, it has an extensive set of Trace commands integrated into the system that let you identify details in the execution sequence of commands. Mathematica's GUI development environment is a toolkit based on Java classes—in particular, on the extensive class libraries of Java GUIs. The GUIKit provides high-level expression syntax for defining common user interfaces, obviating the need for the developer to write Java code. Mathematica comes with MathLink, J/Link, and .NET/Link, all of which let users call to and from code written in C/C++, Java, and .NET languages. Mathematica also comes with a built-in database link.

Matlab. Matlab's standard interactive interface is a window with a partition for processing command-line input. The vehicle for complete application units is either an M-file, consisting of Matlab commands, or a compiled version of an M-file created for a platform-specific target. The Matlab programming language's architecture is such that only relatively simple elements are executable in a single code block within a command-line structure. Creating and testing most computational codes thus involve two separate operations: creating M-file text, and then executing them to test and debug. Matlab provides both a set of debugging commands that can be embedded in M-code and a graphical debugger for fully interactive debugging. Most program developers using standard languages; their code development environments will find the Matlab environment to be familiar. Matlab also provides a development environment for creating specialized GUI's (GUIDE). Moreover, it has a definition for interfacing to external routines written in other languages, including Java, C, and Fortran, as well as to a variety of data objects and servers that communicate via the Component Object Model (COM). Calling C, C++, and Fortran via wrapper functions, VB via COM, and Java classes from Matlab is fully supported and documented. C++ is partially supported, but calling Matlab from C, C++, and Fortran via the engine library and VB via COM is fully supported. Calling of Matlab from Java and C# is partially supported.

The importance of program code management and debugging facilities is directly proportional to the complexity and number of instructional products that the developer intends to create. Among student users, these facilities are most important for computational programming, less important for simulations, and least important for tutorial materials.

Formatting

In the sense we use it here, formatting refers not only to the appearance of text and notational structures, but also to the structural organization of expressions and data. This latter meaning is closely associated with syntax, but we include it here because these packages provide, to varying degrees, tools to expedite command and data entry as well as the text's appearance. A primary difference among the packages is in how they deploy such formatting tools. All three use a consistent concept of textual styles, but their realizations are fairly different. For Maple and Mathematica, access to formatting tools is located mainly in utility palettes, whereas for Matlab, they appear exclusively in pull-down menus.

Maple. Maple has extensive style facilities that couple text to functional and structural elements. It assigns separate, definable styles to input, text output, graphical outputs, and expository paragraphs. If you select a style from a pull-down menu according to function—for instance, "Maple Input"—then that textual style is applied to the subsequent input. Moreover, the kernel treats that input as Maple language. If you select text style, the subsequent input appears (and is treated by the kernel) as expository text. Maple also provides a selection of tools for data and command formatting on palettes in the system development window. You can select alternatively among expression, symbol, matrix, and vector renditions of these palettes; using them can considerably reduce the typing burden for both developer and user in the Worksheet environment. On the output side, as we pointed out earlier, Maple emphasizes the appearance of standard mathematics.

Mathematica. Mathematica's hierarchically structured system of input cells has functional implications for code execution. The kernel distinguishes between Mathematica language and expository text via the key used to terminate an input: the Return key indicates expository text and the Enter key, Mathematica language in the input stream. On the output side, as we've mentioned, Mathematica is the least constrained among the three by mathematical convention. With respect to the appearance of text however, it has a system of predefined styles applied via pull-down menus. Moreover, Mathematica provides an elaborate range of "appearance environments" that anticipate what size and layout choices best satisfy certain operational modes, such as "working," "slide show," and so on. For the developer as well as the user, Mathematica provides customizable tool palettes to implement formatting, simplify command entry, and reduce the typing burden. Mathematica also provides its own specialized wizard functions, among them a set specifically for authoring: `MakeProject`, `MakeIndex`, and `MakeContents`.

Matlab. Matlab places its formatting commands for text in pull-down menus. It permits the author to set style properties for target objects, but if those objects will contain considerable amounts of instructional text rather than, say, labels or legends, the developer must generate the formatting by coding it or creating it exogenously (for example, in a word processor) and importing it. This limited capability is consistent with what programming environments require—elaborate formatting isn't important here. Although importing files into an application is quite easy in Matlab, it makes the preparation and use of considerable amounts of text with sophisticated formatting a comparatively awkward process.

Text formatting is very important for creating tutorial material, less important for simulations, and relatively unimportant for computational programming. Less obvious is the importance of functional formatting—using formatting to distinguish between input and output, to link typography with logical organization, to serve one or another disciplinary custom, and so on. The formatting choices the packages offer seem to match up with their user bases: Maple has a traditional following in the mathematics education community, Matlab in the engineering education community, and Mathematica in the physical science education community. These have as much to do with history as with computational capability, yet the packages' formatting capabilities seem to favor the needs of their communities of educational developers.

Interactive Graphical Output

Interactive graphical output is an important capability that all three packages give the education community. In fact, the integration of features such as interactive, user-rotated 3D displays into powerful yet easy-to-use computational environments is a particularly heroic achievement of these computational productivity packages. The capabilities of their graphical production facilities and interaction tools for casting and recasting output are hugely detailed and beyond this review's scope. However, they're sufficiently important that they are a must for evaluation before adopting a package.

The ability to iterate between modifying the code and manipulating graphical results provides great efficiency for computational science education. In this sense, the graphing capabilities are equally important for simulations, tutorials, and computational programming, and equally so for both developers and users.

Conclusions and Implications

So what does this mean to instructors and students? Our basic conclusion, and the premise in all that follows, is that *all* science and engineering students should have experience in using modern computational tools. This is hardly a radical conclusion. As we've pointed out, undergraduate engineering program accreditation already requires this. Even so, our survey has given us the impression that in undergraduate sciences at least, such a commitment is "more often honored in the breach." But what kinds of computational experiences are adequate and appropriate for all undergraduates? In addressing this question, it might be useful to refer back to the questions we used to frame the first installment of this series:

- To what extent do these tool packages qualify as modern engineering tools?
- What kinds of computational experiences with them are appropriate for undergraduate students?
- What are some major educational goals for science and engineering undergraduates?
- How are specific computing tasks re-

lated to those goals?

- How does each of the three productivity packages realize the required computations?
- How well do these tool packages serve for materials-development work that faculty will likely perform alone?
- How efficient are they when fast response times are required for modifications?
- How expensive are they to purchase and, equally important, maintain?

Our own experiences and those of the academic users we interviewed suggest a few observations that might also be helpful here.

They offer basically the same functional capabilities, but they've evolved from different origins via different paths. Although we have the assurance from our interviewees that, for most instructors, the similarities can make the differences bearable, nonetheless the personality differences can have an impact, as we discovered in our own investigations.

Matlab started from a suite of numerical algorithms and was developed for computational engineering applications. It assumes that the user will need to develop computational skills to work effectively. Its strength lies in programming numerical computations, and it has an engineering look and feel, extending from its development interface's functionality to its workflow organization. It has a very large array of specialty add-on's (or tool boxes), reflecting its large presence in the professional engineering workplace. Its working environment—the same for both developers and users—is very close to that of standard code development systems built for third-generation languages such as C++ and Fortran. It will seem most natural to those who are accustomed to programming and are willing to give up standard mathematical notation and facile interactivity in the standard user interface.

Mathematica, arguably, started from an effort to systematize mathematical computing. It has a unique and rather exotic concept of primitives—in effect, nudging users away from conventional ways of thinking about computing toward a possibly more valuable, yet more demanding, mathematical view. For example, Mathematica's computational primitive is the expression, which it uses to represent everything conceptually—not just mathematical formulas, but also lists and graphics. Mathematica uses the list as a primitive data type, even for matrices. This economy of primitives affects more than just internal implementations of algorithms. As our earlier examples showed, it also influences the external representations of mathematical objects—notations and vocabulary. Mathematica challenges users to think systematically about mathematical computing, albeit within a landscape that might seem somewhat alien to conventional scientists and engineers.

Maple, in contrast, favors mathematical convention. It started from an effort to make symbolic algebraic computing available and accessible to the broad academic community. In its notation and vocabulary, it leverages the user's familiarity with mathematical convention rather than trying to reshape it. This manifests itself in a dominant personality trait—simplification. Nowhere is this more evident than in the way it displays its tools to emphasize their relevant mathematical structures. Maple's vector and matrix construction templates (on the tool palette), for example, distinguish between row and column vectors, and between $(n \times m)$ and $(m \times n)$ matrices. Many shortcuts in its command vocabulary and the implicit assumptions it uses in interpreting input are common to mathematical analyses.

If you're an educator pondering the implications of all of this, a good question to ask yourself is which of these features serve your needs and is most useful for your students for learning? Furthermore, how much time will your students need to achieve the competence to use these things? Given the materials you'll need to prepare for them, how much time and what degree of support will you have for this work? What demands does each package put on you as a developer compared to this support? What's your target audience's level of sophistication? How much will it cost to give them functional access to your materials?

Some recurrent themes that reflect such questions resonated throughout our interviews with experienced users. The steepness of the learning curve for any package depends heavily on past experience. This quickly became evident because we couldn't get a general agreement on which of the three was easiest or most difficult to learn. A clash of personalities—the users' and the packages'—might well be a factor here.

We mentioned GUIs earlier, particularly their relative uniformity from the user's standpoint, but almost no application in our interview sample actually employed specialized GUIs beyond the introductory course level—the relative abundance of examples of applications using them on the three companies' Web sites, notwithstanding. This might have something to do with the relative difficulty in preparing such GUIs and the degree to which they constrain free exploration of the application and its code. But it also could be that our interviewees were long-time users and had started before specialized GUI tools were available, thus they might have neglected them.

Moving applications to the Web is another conundrum. Although the publishing industry widely heralds Web delivery of instructional applications as the wave of the future, our interview sample shows that the future isn't here yet. One reason for turning to the Web for delivery is to disseminate instructional materials expeditiously. This is most immediately useful to your own students, who are increasingly spending time around computers in public campus labs or in their dormitory rooms. But it also carries the promise of deploying your applications across platforms in a transparent way. One possible strategy is to provide at least limited package functionality to students without needing to supply each of them with a copy of the package's kernel.

A detailed discussion of Web features deserves separate review, which puts them beyond this article's scope. However, we can point out the notion of using the Web to deploy computational tools; with the concomitant speed and configuration limitations imposed by the Web environment, it benefits academic applications more than industrial ones. Accordingly, it's not surprising that Web strategies are seemingly pursued most actively by Maple, somewhat less and certainly differently by Mathematica, and least by Matlab. The best recommendation might be for you to see for yourself by visiting their respective Web sites: www.maplesoft.com, www.wolframresearch.com, and www.mathworks.com.

Every undergraduate science, mathematics, and engineering program should consider including a serious computational component in their major curriculum. We believe that any of the three packages we've discussed is capable of supporting such inclusion, but the choice must be made locally. Ultimately, you must consider the detailed mechanics of each package, preferably within the context of other developers' experiences, when selecting a package for any category of educational application (tutorial, simulation, or computation), and the range and power of capabilities as they relate to your local needs.

In our interviews, we discovered several nuggets from experienced users. For one, you reap considerable benefits from settling on a single package, and the wider the scope of commitment (department, area, and institution), the better. With breadth of adoption comes communities of support within student, faculty, and administrative realms along with related pricing benefits. Unfortunately, gaining adoptive consensus can be difficult, but it's important to remember that the choice isn't a lifelong commitment; some interviewees suggested that periodically revisiting adoption commitments is a good idea despite the preferences/prejudices that accumulate with time. Most of us have "accidentally" adopted one or the other of these packages, so we might discover with subsequent evolutionary development that another might better serve our needs. Almost everyone agreed that it was most useful to students that they use a single package for multiple courses over more than a single year. That said, you should consider the differential problems with using a package in introductory as compared to upper-level courses.

A final issue is pricing: because competition is fierce among the companies, this is best left to local and time-dated negotiations, but we can cite a simple metric. "Basic" student access for any package falls near US$100, which is less than the cost of MS Office. What "basic" means varies among the products, but the integrated capability from mathematical computation to graphical output is always at this base. See each package's Web site for an up-to-date price listing and, most important, the licensing terms.

This concludes our experimental review of the instructional use of the three major computational productivity packages. We regret that we couldn't include other comparable products due to space and time limitations, but we hope that the same framework for critical review we've provided will help.

In later issues, we hope to extend this type of review—ostensibly, of the same 3Ms—to their uses in research and in communication settings. If this initial review is to serve as a real experiment, we need data from which to draw information that can shape the style and content of these future review efforts. For this purpose, we've prepared an online survey for readers that will require only a few minutes of your time. We invite you to participate by going to the *CiSE* Web site: www.computer.org/cise.

Norman Chonacky, most recently a senior research scientist in environmental engineering at Columbia, is currently a research fellow in the Center for UN Studies at Yale. Chonacky received a PhD in physics from the University of Wisconsin, Madison. He is a member of the American Association of Physics Teachers (AAPT), the American Physical Society (APS), the IEEE Computer Society, the American Society for Engineering Education (ASEE), and the American Association for the Advancement of Science (AAAS). Contact him at cise-editor@aip.org.

David Winch is emeritus professor of physics at Kalamazoo College. His research interests are focused on educational technologies. Winch received a PhD in physics from Clarkson University. He is coauthor of *Physics: Cinema Classics* (ZTEK, videodisc/CD 1993, DVD/CD 2004). Contact him at winch@taosnet.com.