Graphical User Interfaces in an Engineering Educational Environment

CHRISTOPHER DEPCIK, DENNIS N. ASSANIS

1231 Beal Avenue, 2032 W. E. Lay Automotive Laboratory, University of Michigan, Ann Arbor, Michigan 48109

Received 4 March 2004; accepted 7 September 2004

ABSTRACT: Graphical user interfaces (GUIs) are being increasingly used in the classroom to provide users of computer simulations with a friendly and visual approach to specifying all input parameters and increased configuration flexibility. In this study, the authors first describe a number of software and language options that are available to build GUIs. Subsequently, a comprehensive comparative assessment of possible alternatives is undertaken in the light of a benchmark educational program used in a course on computational fluid dynamics (CFD) at the University of Michigan. For the GUIs presented, their educational value with respect to flexible data entry and post-processing of results has been demonstrated. In addition, the authors offer recommendations for pros and cons of available options in terms of platform independence, ease of programming, facilitation of interaction with students, and flexibility. © 2005 Wiley Periodicals, Inc. Comput Appl Eng Educ 13: 48–59, 2005; Published online in Wiley InterScience (www.interscience.wiley.com); DOI 10.1002/cae.20029

Keywords: graphical user interfaces; classroom; programming; benchmark; visual post-processing

INTRODUCTION

Graphical user interfaces (GUIs) are being increasingly used in the classroom to provide users of computer simulations with a friendly and visual approach to specifying all input parameters, thus making it easier to describe what is needed to run a simulation. While this can be effectively done in a command-line fashion, prompting the user for successive input parameters like DOS programs, a GUI allows the user to see everything at once. Hence, data entry becomes much easier because of the visual aid instead of trying to remember all of the different command-line prompts, or specifying inputs in non-descript, text input files.

The most important benefit of a GUI is that it can *post-process* the results of the simulation providing the user with *instant feedback*. This is especially important when performing parametric studies where variables are changed over a certain range. A visual illustration of how results change as a function of various variables, or parametric sweeps, reinforces the lessons learned in the classroom. While command-line programs can write data to a text-file, this file would have to be opened within another program

Correspondence to C. Depcik (depcikc@umich.edu).

delaying this transmission of knowledge. This instant feedback is particularly valuable when programming and debugging, as something new typically never works right the first time according to Murphy's Law [1]. A GUI may even make it possible to open-up the range of system or cycle configurations that a student can study under one unifying programming environment, versus using multiple, hard-wired codes dedicated codes for studies of each new configuration [2].

GUIs utilize high-level languages to communicate with operating system software, like Windows[1], to provide the user with an aesthetically pleasing interface. GUIs can be written in virtually any programming language and even cross-pollinated with a couple of different languages, in mixed-language programming. It would seem that given a firm grasp of a programming language, it should be straightforward to build a GUI with that language; however, this is not always the case. While a number of studies have compared high-level programming languages and their computational speed [3−5], comparative studies on their potential for creating GUIs are relatively sparse [6−8].

In this study, the authors describe a number of software and language options that are available to build GUIs and undertake a comprehensive comparative assessment of possible alternatives in the light of a benchmark educational program used in a course on computational fluid dynamics (CFD) at the University of Michigan. More specifically, the following languages were studied, with their representative software in parenthesis:

- FORTRAN (Visual Fortran 6.6[2]),
- C/C++ (Visual C++ 6/7[3]),
- Java[4] (Visual J++ 6/7[3]),
- MATLAB[5] (MATLAB R13),
- Basic (Visual Basic 6/7[3]).

The software listed above does not constitute a full list of available options, as other programs such as Metrowerks Codewarrior[6] and Sun Java Studio do exist for compilation of the different languages. However, the GUIs developed by using the above list are representative of those that can be developed by other programs.

[1]Windows® is a registered trademark program of Microsoft Corporation.
[2]Visual Fortran® is a registered trademark program of Compaq Computer Corporation.
[3]Visual Basic®, Visual C++®, and Visual J++® are registered trademark programs of Microsoft Corporation.
[4]Java® is a registered trademark program of Sun Microsystems.
[5]MATLAB® is a registered trademark program of The Mathworks, Inc.
[6]Metrowerks Codewarrior is a registered trademark program of Motorola.

While for a specific application, a certain language may have concrete advantages. It should be emphasized that this assessment is not undertaken with the objective to proclaim the best language for programming GUIs for all classroom applications in engineering. Multiple factors would ultimately enter an individual's choice of the best option, including prior familiarity with an underlying programming language, since it may otherwise be overwhelming to learn a language while also building a GUI at the same time. Instead, this study explores and assesses the pros and cons of different available options to give the reader a solid background for making an informed choice prior to moving into GUI programming for classroom applications.

COMPARISON OF LANGUAGES

To learn about the different GUI programming options, a benchmark program was used. To illustrate the educational nature of GUIs, the program chosen was a driven cavity flow problem used in a course on CFD at the University of Michigan. This problem is illustrated in Figure 1 where the flow inside the box is driven by a plate moving at a constant velocity U. The governing equations of motion for the flow within the box are the incompressible two-dimensional Navier-Stokes equations:

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0 \qquad \text{(continuity)}, \tag{1}$$

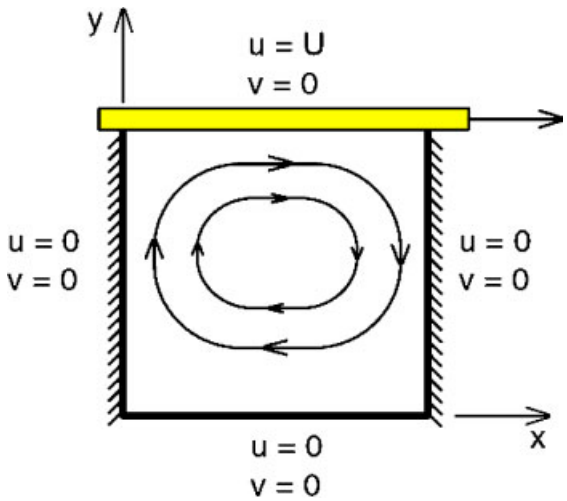

Figure 1 Driven cavity problem used as benchmark to measure the computational efficiency of each lower-level language. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)(\text{x} - \text{momentum}), \tag{2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)(\text{y} - \text{momentum}), \tag{3}$$

where u is the velocity in the x-direction, v is the velocity in the y-direction, ρ is the density, p is the pressure, and ν is the kinematic viscosity. For the incompressible flow Navier-Stokes equations, the energy equation does not need to be solved to find the velocity field unlike the compressible form of the equations. The energy equation *is* a function of the velocity field, but its solution was not important for the purposes presented in this study.

To simplify the numerical calculation, the above three equations of motion can be combined into streamfunction (ψ) and vorticity (ω) format as follows:

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = -\omega, \tag{4}$$

$$\frac{\partial \omega}{\partial t} + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} = \nu\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right), \tag{5}$$

The solution of the above equations occurs via the Successive Overrelaxation (SOR) method [9], which is used to increase the convergence rate of an iterative method by propagating the corrections ($\Delta\psi^n = \psi^{n+1} - \psi^n$) faster throughout the mesh:

$$\psi_{i,j}^{n+1} = \frac{\beta}{4}(\psi_{i+1,j}^{n} + \psi_{i-1,j}^{n+1} + \psi_{i,j-1}^{n+1} + h^2\omega_{i,j}^{n}) + (1-\beta)\psi_{i,j}^{n} \tag{6}$$

where $1 < \beta < 2$,

$$\omega_{i,j}^{n+1} = \frac{1}{h}\left(v_{i+1,j}^{n} - v_{i,j}^{n} + u_{i,j}^{n} - u_{i,j+1}^{n}\right), \tag{7}$$

with h equal to the discretization in the x- and y-direction for a symmetric grid (side length $= L$). The time-step for the simulation is calculated as:

$$\Delta t \leq \frac{h^2}{4\nu}. \tag{8}$$

Since this problem is quite computationally intensive, it provides for a good comparison of the run-times of the different programming languages. As with most engineering problems, a premium is often put on fast run-time for quick turnaround of results. In this study, the computational efficiency of FORTRAN and C/C++, often used by the engineering community, is compared to Java and MATLAB. It is often a good idea to perform self benchmarking because you will be able to understand your own limitations in regards to programming efficiency and compiler use [10].

The main benefit of using this problem is that it provides a nice graphical description of the results when plotting the streamfunction and vorticity (evident in later figures). This is invaluable because when programming the simulation the students immediately know if they have coded it correctly. In addition, it will help inform the students when they choose the wrong parameters. For instance, if they do not calculate the time-step according to Equation (8), the wrong solution will be immediately displayed. Therefore, incorporating the streamfunction and vorticity graphs within our GUI will reinforce important CFD lessons.

Grid sizes of 10 × 10, 20 × 20, and 40 × 40 were used to solve the problem and generate run-times for the different languages. The run-time was calculated by using the clock function in the respective language for the amount of time taken in *only* the computationally intensive incompressible 2-D N-S solver. Table 1 gives the different time-step and number of steps needed for convergence of the simulation based on grid size. Table 2 has the results of the computational run-time of the different languages as a function of the grid size. The computer used ran Windows XP Professional and had a Pentium 4 processor running at 2.26 GHz with 1.00 GB of RAM.

The results of the computational run-time show that the fastest language was C/C++ whereas the slowest was MATLAB. For the 10 × 10 grids, both FORTRAN and C/C++ ran so fast that there was no difference between the clock settings; hence the not available results in the table. The results for MATLAB are to be expected because of the fact that its code is not compiled before running unlike all the other languages. Instead, the user creates a text file with the MATLAB code which the program then runs as a script.

Release 13 of MATLAB added the option of converting the code into C or C++ (*mcc*), which is

Table 1 Run Conditions for the Driven Cavity Flow Problem That Ensure an Equilibrium Final Condition ($\boldsymbol{\beta} = 1.5$, $\boldsymbol{\nu} = 0.1$, $L = 1$).

Grid size	Time-step	Number of steps
10 × 10	0.01543	1,000
20 × 20	0.00346	1,500
40 × 40	0.00050	3,000

Table 2 Computational Run-Time in Seconds of Each Different Programming Language for the Driven Cavity Flow Problem With Standard Deviation in Parenthesis

Language	10×10	20×20	40×40
C/C++	N/A	0.2684 (0.0060)	20.36 (0.03)
FORTRAN	N/A	0.2912 (0.0075)	22.88 (0.06)
Java	0.0156 (0.0005)	0.4401 (0.0073)	31.12 (0.30)
Basic	0.0345 (0.0063)	1.1940 (0.0166)	86.51 (0.18)
MATLAB	0.3931 (0.0850)	6.9953 (1.5483)	368.33 (20.07)
mcc	1.4035 (0.0108)	51.619 (0.411)	3681.44 (8.74)

then compiled with a dedicated C/C++ compiler. When performing this action within MATLAB, the run-time actually *increased* even though the same C/C++ compiler was used as with the dedicated C/C++ code. This is a known bug with the MATLAB Compiler 3.0 used within MATLAB R13 [11]. It is interesting to see that creating the C/C++ puts a lot of overhead into the code increasing the total number of lines along with the complexity. In the newest version of MATLAB, R14 not yet analyzed by the authors, the release notes state that this C/C++ feature has been eliminated and there is no speed difference between a compiled application and running it in MATLAB. To speed up the code, there is an option within MATLAB, called the Just-In-Time Accelerator, which can be used to analyze the code to find ways to optimize its computational efficiency.

If a run-time result is counterintuitive to the reader's personal experience, maybe that C/C++ is faster than FORTRAN, it may be because of the compiler, optimization options, and/or programming technique. The authors are sure that people can write code that runs faster than the above, but that was not the point. This study is meant to help explain how well *a single engineer* can write and build GUIs with a number of different languages. By comparing code from a number of different sources, the results would not be as instructive.

IMPORTANT CONCEPTS

Before describing the GUIs, a few important concepts need to be explained to help the reader understand options present in the different programs. These concepts give the reader the ability to further understand the ease and difficulty of programming GUIs given their own background.

Mixed-Language Issues

From the above comparisons, it may not be beneficial to build a GUI completely in a language, like MATLAB or Basic, due to their relatively slow run-times. As a result, it would be advantageous to be able to mix languages, like one for the GUI and the other for the computational code. The only instance of perfect compatibility found by the authors comes with Visual Fortran 6.6 and Visual C++ 6, where FORTRAN and C++ code can be compiled *at the same time* when building a GUI; providing both programs are installed[7]. This compilation was done by simply adjusting a number of compiler settings.

To use mixed-language programs with the other software programs certain middleman files need to be built. On a Windows environment, these are dynamically linked libraries (*dll*). Essentially these are *external subroutines* instead of the internal subroutines or functions typically used by programmers. They are placed in the same directory as the executable program (the actual GUI program) and are called in virtually the same way as internal subroutines. However, they have a specific format in the function call header that is needed to utilize them. In a Visual Basic example that follows in a later section, the authors have two versions. In one version, the GUI built-in Visual Basic calls a dll written in FORTRAN.

MATLAB is slightly different from the facet that the dll is actually built within the MATLAB program. In this case, they call it a *mex* file and to use this feature, MATLAB has to be setup by the user to find the compiler that will be used to create the mex file. So, instead of compiling a dll separately with the software of choice, MATLAB will do this for you providing you tell them which compiler to use [12].

Direct Calls Versus API Libraries

There are two ways of writing GUI code, one is the brute force method (direct calls) and the other is more elegant (API libraries). In the direct call method, the user calls the Windows functions that are written in the respective software language. In this approach, the programmer is responsible for ''mapping'' all the

[7]Visual Fortran 6.6 does not support Visual C++ 7 (or .NET); however, Intel Visual Fortran 8® (a trademark program of the Intel Corporation) is supposed to be the replacement for Visual Fortran 6.6.

windows calls to and from the program. This means that the programmer has to account for all mouse-movements, keyboard pressings, calls from other windows that are open, etc.... From the authors' experience, this is a complicated way of building GUIs and can result in crashing the code (and Windows) if not careful.

An API, or Application Programming Interfaces, library makes up an ''application framework'' on which the programmer builds an application. At a very general level, the framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton [13]. The user then has to fill in the rest of this skeleton with simulation specific items. This library alleviates the burden on the programmer because it has already done a lot of the ''mapping.'' It is a more graceful way of building code, however, based on the programming language as explained later, this mapping may be easy to learn or somewhat complicated.

Platform Independence

Java is truly the only programming language that provides complete architecture neutrality. It is an interpreted language, so it can run on any platform that has the Java interpreter and run-time environment. The Java interpreter translates compiled bytecode, so a compiler is still used to create a compiled code. But this compiled code can be read on any machine with the Java interpreter (one compiled code, many machines). At this point, the reader might think, why bother with all the other programs then, here is one that everyone can use. However, there are three issues with this.

The first issue is that Java programs run slower than FORTRAN or C/C++ programs as seen above and explained in [14]. However, this is a function of the compiler, so eventually Java might become as fast as the more established languages.

The second issue is that Java is relatively new, so most existing code would have to be rewritten in Java. There are separate programs that can change your code from one language to another similar to the functionality presented in MATLAB. A prime example of this is f2c, which converts FORTRAN to C [15]. However, while these programs generally work, the converted code in the new language can be very difficult to read and not well organized. This was experienced with the MATLAB conversion in an earlier section. Since Java is completely an Object-Oriented Programming (OOP) language, this conversion is sure to be messy. OOP forces the user to confine themselves to certain standards (classes and inheritance), limiting some of the freedom a user might be accustomed to (ex: global variables not allowed). It is possible to link with external subroutines through a Java Native Interface (JNI); however, these codes would then become platform dependent. For example, dlls created in Windows do not work on a Macintosh platform.

The last problem involves post-processing the data. In the case of all GUIs that were built in this study, a third-party program was incorporated into the GUI language to provide the graphing capabilities. This was done because the authors did not want to write their own graphing software and have to make it aesthetically pleasing, which can require a large amount of excess effort. Instead, the compiled code calls an API that encompasses the graphing calls very neatly. This graphing software is operating system dependent and can only be installed on a Windows machine. To provide for platform independence, the user would have to create (or find) a Java graphing package.

There are some software programs that can create GUIs for multiple platforms. Metrowerks Codewarrior does allow for Windows and Macintosh programs to be created at the same time using C/C++ code. However, this code must be written using the direct call method. REALbasic[8] for the Macintosh has the ability to import Visual Basic programs through a program called VBCleaner. While untried by the authors, previous experience with converters like f2c and mcc creates some skepticism of the ability to cleanly port this code to the other platform.

GUI BUILDS

In this section, GUIs are built using the different languages previously mentioned. The GUIs built are of the simple form-based type where all input and output are shown in a single window. These GUIs are similar to the calculator program that comes with the Windows operating system. All GUIs can run as stand-alone (executable) applications easily installed on other Windows machines.

Visual Fortran

To create the GUI with Visual Fortran shown in Figure 2, a palette is given that allows the user to place the representative entry fields (ex: X Grid Points) and buttons (ex: RUN) on a blank form. While this is quite easy to do, implementing the functionality behind the code is much harder. This is the ability of the code to grab the data from the GUI and implement it in the

[8]REALbasic is a registered trademark program of Real Software, Inc.

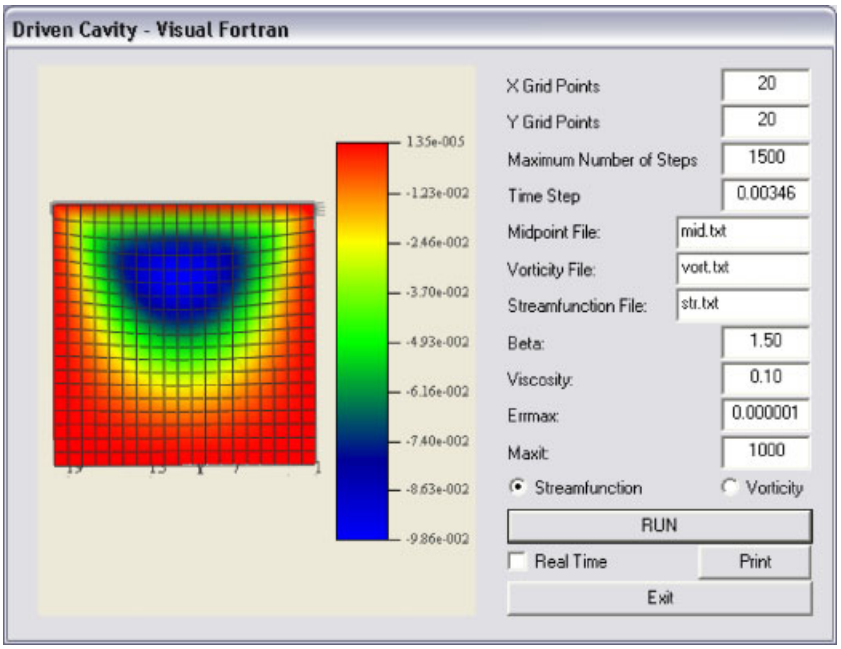

Figure 2 Driven cavity flow GUI created with Visual Fortran (streamfunction shown). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

N-S solver when the RUN button is pushed. In Visual Fortran, the code has to be written in the direct call method, which is the most difficult to understand and program correctly. With the lack of assistance outside the online help and sample code provided with the program, this can create a lot of chaos for the programmer.

Visual Basic

Visual Basic has previously been used by engineers to create customized software [16–18]. In these studies, the authors explain the increased visual aspect of GUIs and their advantages in engineering. This is particular important for this study due to the driven cavity flow problem and its numerical solution.

Creating the GUI with Visual Basic shown in Figure 3 is as easy as the Visual Fortran program. A palette is again given thast allows the user to place the representative entry fields and buttons on a blank form. However, unlike Visual Fortran, Visual Basic incorporates a number of APIs, which makes it easy to incorporate the logic behind the GUI. It is also important to note that there is a plethora of books available in building GUIs using Visual Basic ([5] was the one used by the authors) along with ample online examples and help that come with the program.

To demonstrate the ability to create a mixed-language program with Visual Basic, the authors created a separate Visual Basic GUI where the computational intensive code written as FORTRAN was incorporated as a dll. This was relatively easy to do and to access the FORTRAN code within Visual Basic; only one line of code was needed. In all of the GUIs written in their native languages, the ability to see the graphs change in real-time was included. However when using a dll, this is not possible because to update the graph in real-time, the program would have to exit the dll to update the GUI and then re-enter the dll to continue the operation. This would require some creative structuring of the dll to implement this feature.

Visual C++

Visual C++ has been used previously to create GUIs for engineering applications [19,20,21]. In this section, its use is documented in the creation of a two similar GUIs; one utilizing the direct call method and the other using APIs in the form of Microsoft Foundation Classes (MFC). In the direct call method shown in Figure 4; all GUI fields had to be written out explicitly telling Windows where to place the field, how big to make it, and what it should look like. This increases the level of complexity and makes it harder to build an aesthetically pleasing GUI. In addition, similar to Visual Fortran, all functionality of the GUIs has to be written by the programmer through the direct call method [22].

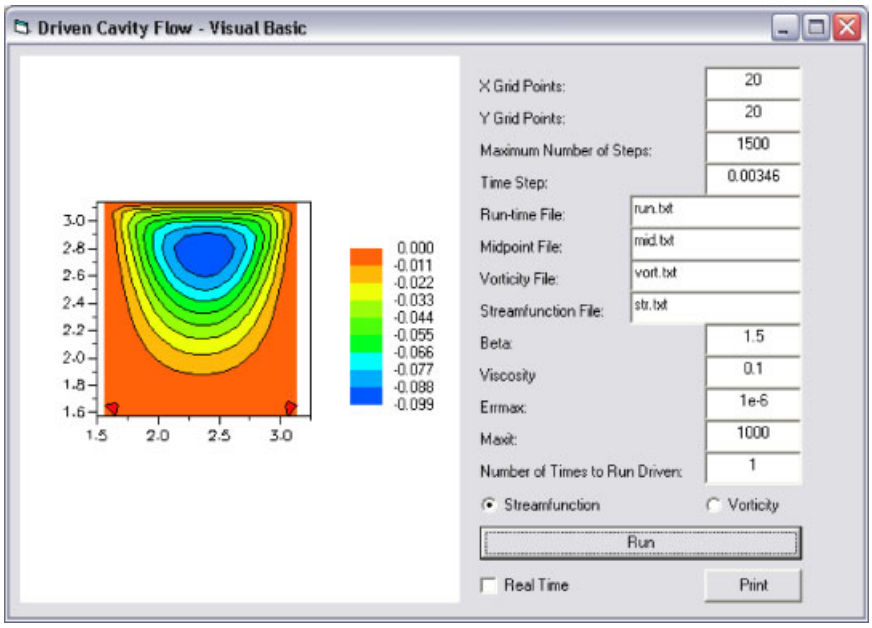

Figure 3 Driven cavity flow GUI created with Visual Basic (streamfunction shown). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

Using the MFC approach, creating the GUI in Figure 5 is exactly the same as the Visual Fortran method. In fact, Visual C++ and Visual Fortran programs both use the same Integrated Development Environment (IDE) as part of a bigger program called Visual Studio[9]. This explains the ability to compile FORTRAN and C/C++ code at the same time as previously mentioned. However, unlike the API use within Visual Basic, the logic behind MFC is of the Document-View architecture (one Document controlling many Views). This is best explained from the fact that in the Visual Basic program there is only one file that the user needs to control, whereas in the Visual C++ program there are many (more than six) that the user must understand to truly allow a fully featured program. There are a number of books written on the MFC approach ([23] was used by the authors), however, it is more complicated than the Visual Basic approach. From the authors' experience, learning this approach can take a significant amount of time. Yet if the reader wishes to build a program with multiple Windows incorporating all the bells and whistles that Windows has to offer, the MFC approach is what is typically used. This can be seen in the first author's doctoral dissertation where a catalyst model was built using Visual C++ [24].

[9]In the newest version of Visual Studio (7 or .NET), Visual Basic, and Visual J++ have been included with Visual C++ in the IDE.

In Figure 6, the authors demonstrate how the GUIs can be used for instant feedback of incorrect input parameters. In this case, the time-step entered is higher than the minimum allowed according to Equation (8) causing the solution to numerically disintegrate.

Visual J++

Typical use of Java is to create applets for web-based applications and even for web-based tutorials [25]. In this section, a stand-alone GUI application is created in Java using Visual J++. Creating the GUI shown in Figure 7, is as easy as Visual Basic and Visual C++ (MFC), but incorporating the logic behind the GUI is slightly more complicated than the Visual Basic example because of the OOP nature of the Java language. However, it is easier than building the GUI in Visual C++. Most of the Windows messaging is built-in and the rest is not difficult to implement (follows along same lines as Visual Basic). Unlike Visual Basic and Visual C++, there is little help available for Visual J++ ([26] used by the authors), because of its relatively new status on the marketplace.

In Figure 7, the vorticity is shown instead of the streamfunction, and the graph is altered slightly to demonstrate that with the third-party graphing software used, the user has the ability to change the graph while running the simulation. This feature can help the student with a further understanding of the results.

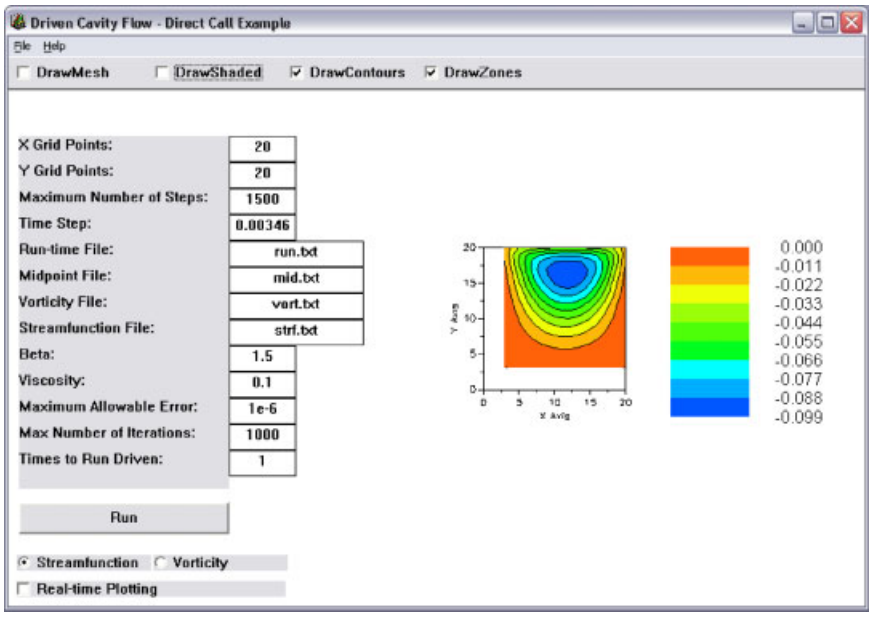

Figure 4 Driven cavity flow GUI created with Visual C++ utilizing the direct calling method (streamfunction shown). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

MATLAB

Several studies have described MATLAB GUIs as a platform for academic research and education [27,28]. To help build a GUI, MATLAB provides a manual aptly named GUIDE (Graphical User Interface Development Environment) [29]. This manual gives the programmer an idea of how to properly design the

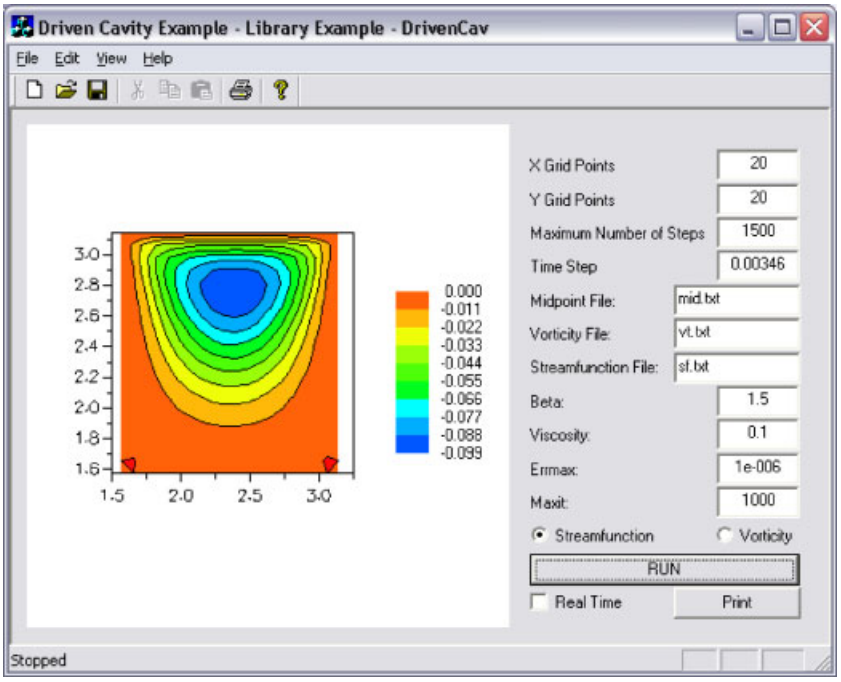

Figure 5 Driven cavity flow GUI created with Visual C++ utilizing the MFC libraries (streamfunction shown). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

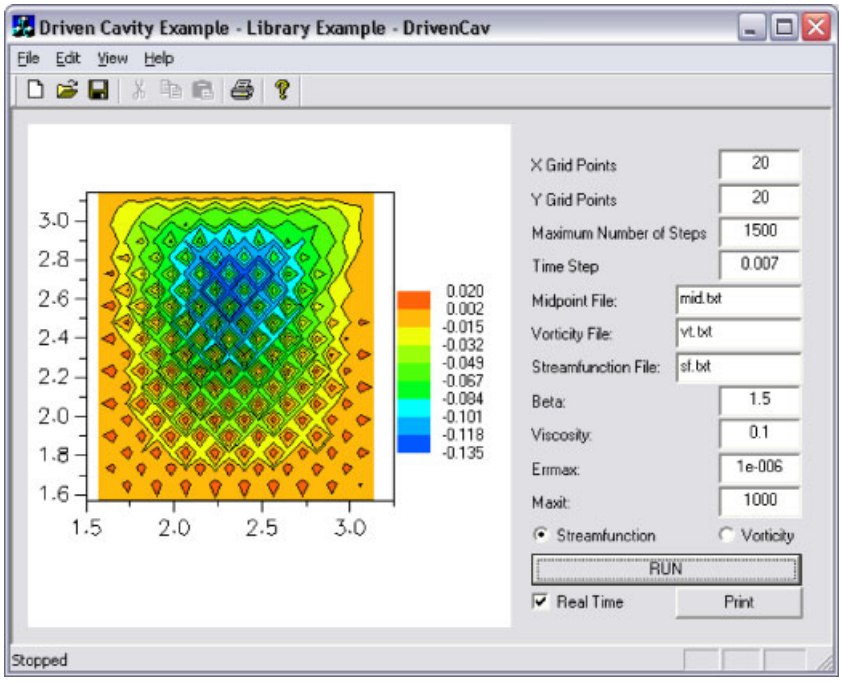

Figure 6 Illustration of incorrect input data (time-step) used in solving incompressible 2-D Navier-Stokes simulation (streamfunction shown). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

GUI, but lacks some information on the calls needed for implementation of the GUI features. Little information on building GUIs with MATLAB exists outside the program ([30] is one resource found by the authors), unlike Visual C++ and Visual Basic.

Using only the MATLAB manual, building the GUI shown in Figure 8 and implementing the code behind it was difficult. Most of the Windows message handling has been incorporated, but calls to and from the GUI, such as what happens when selecting an item

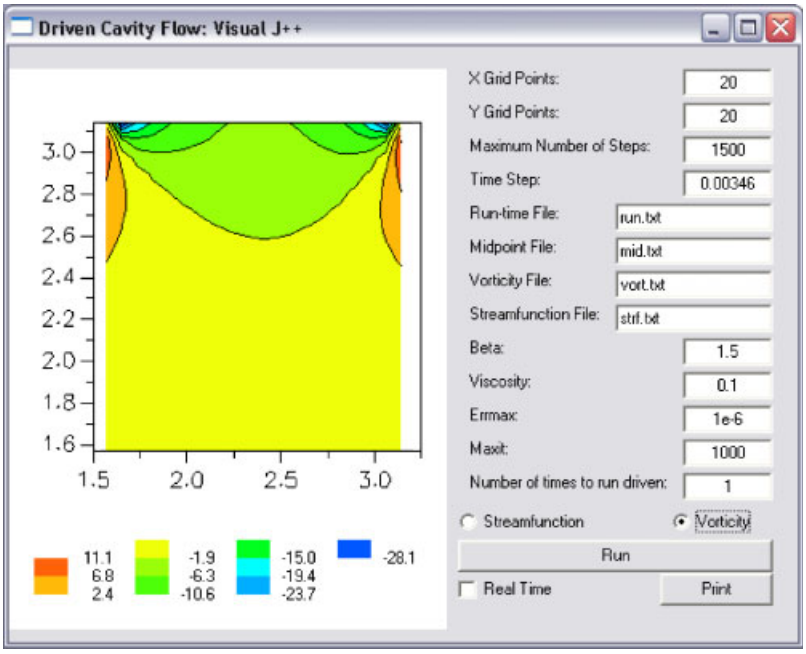

Figure 7 Driven cavity flow GUI created with Visual J++ (vorticity shown). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

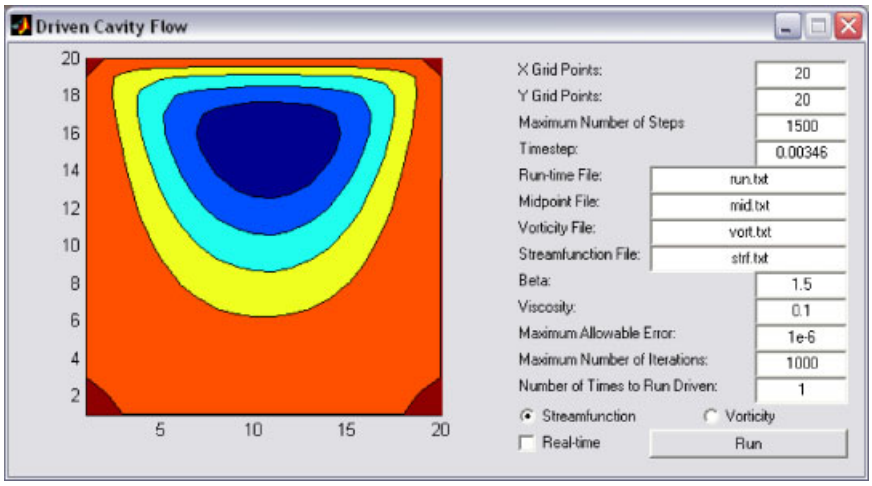

Figure 8 Driven cavity flow GUI created with MATLAB (streamfunction shown). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

from a list, still need to be handled by the programmer. This is where the lack of available references caused a problem.

A great advantage for MATLAB is that it already has all of the graphing features needed. Instead of having to incorporate a third-party program, all post-processing of the data can be done directly by MATLAB. MATLAB R13 also allows the user to create stand alone versions of the GUIs using C/C++ Math and Graphics Libraries. When doing so, it was found that the code will run slower resulting from the same computational bug mentioned in the Comparison of Languages section. MATLAB R14 uses a new feature, named MATLAB Component Run-time (MCR), to build stand alone programs that enable the execution of compiled M-files instead of creating and compiling C/C++ files.

If a user wished to distribute proprietary MATLAB code, the mex function previously mentioned can be used to "hide" this code. In this case, the GUI created in MATLAB would utilize the functionality of mex files instead of the traditional text-based input files. The user also has the option of creating preparsed code (pcode) that hides the proprietary algorithms.

CONCLUSIONS

In this study, the authors described a number of software and language options that are available to build GUIs and undertook a comprehensive comparative assessment of possible alternatives. For the GUIs presented, their educational value with respect to flexible data entry and post-processing of results has been demonstrated. Clearly, for a specific application, a certain language may have concrete advantages and multiple factors would ultimately enter an individual's choice of the best option. Nevertheless, our study can provide valuable insight to the reader for making an informed choice prior to moving into GUI programming for classroom applications. In particular, the following findings can be useful in this context:

- If platform independence is the main goal, then Java should be used as the interface software as well as the underlying programming language. However, to get the full benefit of a GUI, the programmer would have to find compatible plotting software that is also platform independent.
- If someone desires to quickly build a GUI and computational speed is not an issue, then Visual Basic seems to be the best software choice. However, if the simulation to be built is anticipated to be numerically expensive, it would be best to include dlls built in C++ or FORTRAN for the guts of the underlying code. This is also relevant if a large library of existing code is already available. However, the real-time plotting ability of the simulation could be lost in translation.
- If a professor wishes to collaborate with students on a GUI, MATLAB might be the best choice. Since engineering students are typically familiar with MATLAB, subroutines written by the students can be easily checked with MATLAB's graphing capabilities. The GUI can then call the students subroutines without having to worry about compiling and linking the separate codes.
- If a Windows simulation is needed with multiple windows and lots of flexibility, it would be best

to learn the MFC and build the GUI with Visual C++. This is by far the most powerful method for building GUIs, but it comes at the cost of increased complexity. With Visual Fortran, the user also gets the benefit of true mixed-language programming.

All GUIs developed in this study are available to the reader. For more information, please contact the communicating author at depcikc@umich.edu. Using this study and looking at the GUI codes created can help the reader make an educated decision about how to begin GUI programming.

REFERENCES

[1] Air Force Flight Test Center History Office, Murphy's law was invented here, www.edwards.af.mil/history/docs_html/tidbits/Murphy's_law.html, 2004.

[2] C. Depcik, Open-ended thermodynamic cycle simulation, M.S. thesis, Mechanical Engineering, University of Michigan, Ann Arbor, 2000.

[3] D. A. Wheeler, Ada, C, C++, and Java vs. the Steelman, www.adahome.com/History/Steelman/steeltab.htm, 1996.

[4] J. R. Cary, S. G. Shasharina, J. C. Cummings, J. V. W. Reynders, and P. J. Hinker, Comparison of C++ and Fortran 90 for object-oriented scientific programming, Comput Phys Commun 105 (1997), 20−36.

[5] B. Mösli, A comparison of C++, Fortran 90 and Oberon-2 for scientific programming, Proceedings of GISI 95, pp 740−748, Zürich, September 18−20, 1995.

[6] J. Surveyer, Java and Visual Basic: Programming's new breed, www.hubcanada.com/story_2269_22, 1999.

[7] F. Balena, Programming Microsoft Visual Basic 6.0, Microsoft Press, Washington, 1999.

[8] A. M. Al-Rawi, Comparison between high level languages for power engineering education, Proceedings of the 29th Universities Power Engineering Conference, pp 894−897, Galway, September 1994.

[9] C. Hirsch, Numerical computation of internal and external flows, Vol. 1: Fundamentals of Numerical Discretization, Wiley, Chichester, 1988.

[10] R. P. Weicker, A detailed look at some popular benchmarks, Parallel Comput 17 (1991), 1153−1172.

[11] The MathWorks, Inc., Solution Number: 1-19XG3, http://www.mathworks.com/support/solutions/data/1-19XG3.html, The MathWorks, Nantick, MA, 2004.

[12] The MathWorks, Inc., Application program interface guide, The MathWorks, Nantick, MA, 2001.

[13] Microsoft Corporation, Using the classes to write applications for windows, msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/html/_core_using_the_classes_to_write_applications_for_windows.asp, 2004.

[14] J. R. Jackson and A. L. McClellan, Java by example, Sun Microsystems Press, California, 1999.

[15] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, A Fortran-to-C converter, AT&T Bell Laboratories, Computing Science Technical Report No. 149, 1995.

[16] K. Ng, W. M. Presz, Jr., and M. Khosrowjerdi, Engineering analysis and simulation using Visual Basic, Proceedings of 1994 ASME International CIE Conference, pp 897−912, Minneapolis, September 11−14, 1994.

[17] A. B. Savery, Creating Windows based engineering software applications using Microsoft Visual Basic, Comput Civil Eng 1 (1995), 99−106.

[18] D. E. Torres and J. L. Anders, Using Microsoft Visual Basic to write engineering applications, Proceedings of the Petroleum Computer Conference, Society of Petroleum Engineers, pp 291−299, Houston, June 11−14, 1995.

[19] W. L. Whipple, Walking through an application with Visual C++, Proceedings of the ELECTRO'96, IEEE, pp 359−364, Somerset, April 30−May 2, 1996.

[20] Y. Tao, Design patterns for developing GUI applications, Proc Front Educ Conf 1 (1999), 11a3-11.

[21] G. Antonio, R. Fiutem, E. Merlo, and P. Tonella, Application and user interface migration from Basic to Visual C++, Proceedings of the 1995 ISEE International Conference on Software Maintenance, IEEE, pp 76−85, Opio, October 17−20, 1995.

[22] C. Petzold, Programming Windows, the definitive guide to the win32 API, Microsoft Press, Redmond, WA, 1998.

[23] D. J. Kruglinski, S. Wingo, and G. Shepherd, Programming Visual C++, Microsoft Press, Redmond, WA, 1998.

[24] C. Depcik, Modeling reacting gases and after treatment devices for internal combustion engines, Mechanical Engineering, University of Michigan, Ann Arbor, 2003.

[25] D. Cole, R. Wainwright, and D. Schoenefeld, Using Java to develop web based tutorials, Proceedings of the 29th ACM SIGCSE Technical Symposium on Computer Science Education, ITiCSE, pp 92−96, Atlanta, February 26−March 1, 1998.

[26] S. R. Davis, Programming Microsoft Visual J++ 6.0, Microsoft Press, Redmond, WA, 1999.

[27] S. Manson, C. DeMarco, R. Lasseter, and F. Alvarado, MATLAB GUI power flow, Proceedings of the 31st Universities Power Engineering Conference, pp 646−649, Iraklio, September 18−20, 1996.

[28] A. Azemi and E. E. Yaz, Using graphical user interface capabilities of MATLAB in advanced engineering courses, The 38th IEEE Conference on Decision and Control (CDC), IEEE, pp 359−363, Phoenix, December 7−10, 1999.

[29] The MathWorks, Inc., Creating graphical user interfaces, The MathWorks, Nantick, MA, 2002.

[30] P. Marchand, Graphics and GUIs with MATLAB, CRC Press, Boca Raton, FL, 1999.

BIOGRAPHIES

Christopher Depcik received his BS degree in mechanical engineering from the University of Florida in 1997. He obtained his first MS degree in mechanical engineering in 1999 and his second MS degree in aerospace engineering in 2002 from the University of Michigan. In 2003, he obtained his PhD degree in mechanical engineering from the same university under the supervision of Prof. Dennis Assanis. He is currently a research fellow in the Department of Mechanical Engineering at the University of Michigan. His areas of research involve variable-property reacting-gas dynamics and their application to the exhaust of internal combustion engines, including after-treatment devices along with fuel cell simulation. In addition, he is actively involved in developing educational software for engineering courses.

Dennis Assanis is a professor and chair of mechanical engineering and the Jon R. and Beverly S. Holt Professor of Engineering at the University of Michigan, where he is also the director of the Automotive Research Center. He received his BSc degree in marine engineering from the University of Newcastle-upon-Tyne, UK, in 1980. He has received four graduate degrees from the Massachusetts Institute of Technology: SM in naval architecture and marine engineering (1982), SM in mechanical engineering (1982), PhD in power and propulsion (1985), and SM in management (1986). Prior to joining the University of Michigan in 1994, he was an assistant professor (1985–1990) and an associate professor (1990–1994) in the Department of Mechanical Engineering at the University of Illinois Urbana–Champaign. His research interests include modeling and computer simulation of internal combustion engine processes and systems; experimental studies of engine heat transfer, combustion, and emissions; and automotive systems design optimization. He has published over 150 articles in journals and refereed conference proceedings, and he is a fellow of the Society of Automotive Engineers.