Int. J. Engng Ed. Vol. 17, No. 6, pp. 580–587, 2001
Printed in Great Britain.

0949-149X/91 $3.00+0.00
© 2001 TEMPUS Publications.

A Self-Checking Interface For MATLAB-Based Interactive Exercises*

BRIAN L. F. DAKU
University of Saskatchewan, Department of Electrical Engineering, 57 Campus Drive, Saskatoon, Saskatchewan, S7N 5A9, Canada. E-mail: daku@engr.usask.ca

Effective computer-based educational learning tools must engage the student with significant questions to deepen understanding of the subject. These tools should also provide immediate feedback to the student and, if necessary, direct the student into remedial work. This paper describes the construction of a self-checking interface that can be used to evaluate student answers to challenging questions in an interactive computer-based educational tool. This self-checking interface has been implemented using MATLAB. An example exercise, taken from a computer-based tutorial that uses this self-checking interface, demonstrates how the interface operates.

INTRODUCTION

THE DELIVERY of educational material using computer-based methods is being explored in a number of areas [1–4]. Typically, this computer-based approach is used as a presentation tool, with little or no reinforcement of the material through exercises. If a computer-based educational tool is to be effective, the student should answer questions to reinforce the material covered. Immediate feedback should also be provided to maintain the student's interest. An example of a feedback approach to reinforce concepts using multiple-choice questions has been proposed [5]. If the student is to obtain a deeper understanding of the material, more complex and significant problems must be solved. This, of course, makes the task of evaluation and providing feedback more complex.

Solving a more challenging problem requires the use of a computer-based math tool. One example of such a tool, used to solve scientific, financial, engineering or math problems, is MATLAB [6]. MATLAB is a computer programming environment that is used to perform and visualize numerical computations. MATLAB's ease of use relative to traditional programming languages has made it very popular in academic and industrial environments. An example of its popularity is the large number of textbooks that make use of MATLAB [7]. Thus MATLAB appears to be a good choice for integration with interactive computer-based educational products.

MATLAB provides the student with an ideal tool for solving complex and significant reinforcement problems. In a computer-based educational product, there must also be a way of evaluating the solutions to these problems to provide feedback to the student. One way this can be done is to use MATLAB in conjunction with the system program to evaluate the solution. This is an ideal approach, with little extra overheads if MATLAB is already being used to solve the problem.

This paper describes a unique system, referred to as a self-checking interface, that uses MATLAB to evaluate student answers to challenging questions in an interactive computer-based educational product. The student uses MATLAB to solve reinforcement problems and then MATLAB is used by the system to evaluate the student solutions. The paper presents the approach used and illuminates what one needs to consider when constructing a self-checking interface using MATLAB. An example is presented demonstrating how the self-checking interface could be used in a computer-based tutorial. This example has been taken from a MATLAB tutorial that uses the self-checking interface. This tutorial, *M-Tutor: An Introduction to MATLAB*, has been published as a CD-ROM by Prentice-Hall [8], and a further description of the tutorial is available in articles by Daku and Jeffrey [9,10] or at the website [11].

SYSTEM DESCRIPTION

A computer-based educational product is a software program that runs on a computer. This program can be generated using programming languages such as C, or it can be produced using an authoring package [12]. In this paper, the main software program which implements the computer-based system will be referred to as the Computer-based System Program (CSP).

Extending a computer program that delivers the educational material, from a presentation tool to one with interactive MATLAB exercises, is relatively straightforward, since MathWorks, the company that produces MATLAB, has provided a number of interface routines. These

* Accepted 15 March 2001.

interface routines can be used to directly access the MATLAB computational engine through the operating system, without using the standard command window interface.

The basic structure of the proposed exercise interface, using the MATLAB engine, is shown in the system block diagram of Fig. 1. The Student Exercise window provides a physical interface for the student to solve problems using MATLAB. The MATLAB commands, entered in the Exercise window by the student, are executed by the MATLAB Engine. The MATLAB command execution is performed using custom Dynamic Linked Library (DLL) routines within the MATLAB Engine Interface block of Fig. 1. These DLL routines use a built-in access feature that allows MATLAB to be used within other computer-based systems [13].

The evaluation of the student solution is performed by the Self-Checking Module, shown in Fig. 1. This module consists of a number of MATLAB functions that are used to compare the student solution to the correct solution. These MATLAB functions are executed by the MATLAB Engine, using the MATLAB Engine Interface routines.

A detailed system block diagram, focusing on the Self-Checking Module and the MATLAB Engine Interface is shown in Fig. 2. The MATLAB workspace, shown at the bottom of the figure, contains a set of Student Variables that are the student solution to the problem. These variables are produced by the student by solving the problem in the Student Exercise window. The workspace also contains another set of variables, referred to as Solution Variables. These are the pre-calculated solution variables that are loaded into the workspace by a Self-Checking Module function.

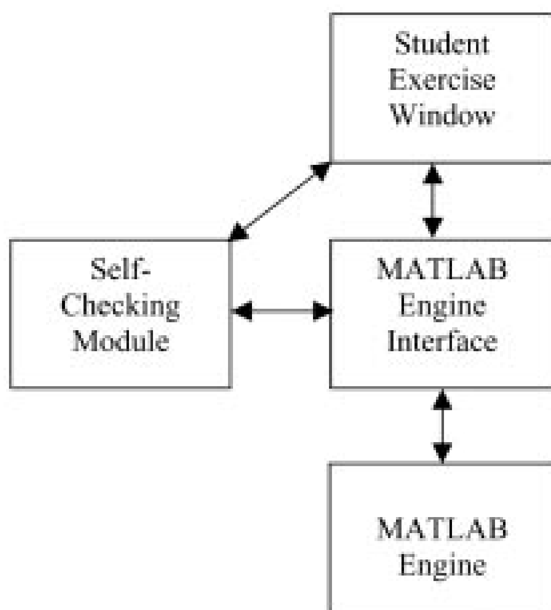

Fig. 1. Exercise interface system block diagram.

The MATLAB Engine Interface block contains a number of custom DLL routines that provide the background interface to the MATLAB engine. These routines will be described in the section 'MATLAB Engine Interface'.

The Self-Checking Module block, at the top of Fig. 2, contains a number of MATLAB functions that are used to compare the workspace Solution Variables with the Student Variables to determine if the student solution is correct. These functions are executed using the DLLs in the MATLAB Engine Interface to access the MATLAB Engine. The Self-Checking Module is described in more detail in the following subsection.

Self-Checking module

The Self-Checking Module contains a number of MATLAB functions that are used to compare variables in the MATLAB workspace. Basically, the comparison functions are used to check the student solution to an exercise to determine if it is correct. There are three basic types of comparison functions: one to evaluate text-based solutions, one for graphic-based solutions and one to permit custom functions for non-standard solutions. A precompiled exercise database is used to determine the type of solution for the present exercise.

This paper will only describe the comparison function used for text-based solutions, since this function is relatively short and easy to understand and thus will provide the best insight into the Self-Checking Interface. The text-based function is called *EvalText* and a pseudocode representation of the *EvalText* function is given in Algorithm 1.

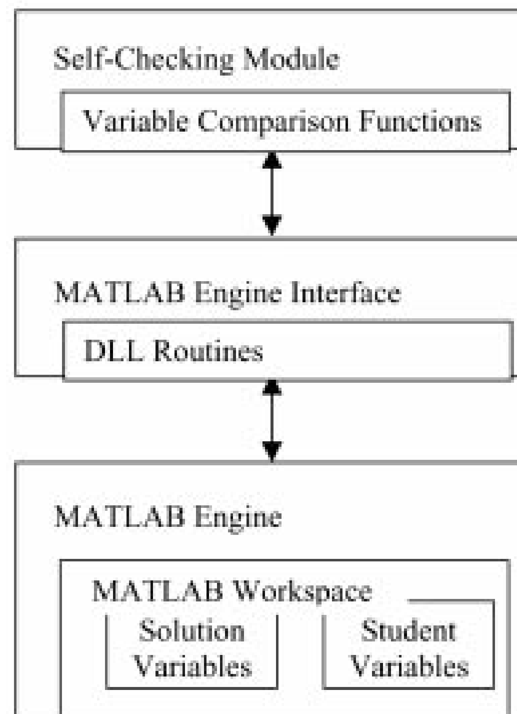

Fig. 2. Detailed system block diagram.

Algorithm 1 EvalText( )

comment: This MATLAB function is used to evaluate exercises that produce alphanumerical text-based solutions.

function *correct* = EvalText

comment: Determine student-generated information.

VarNames ← The names of all student-generated variables in workspace
NumVars ← Number of variables in *VarNames*

comment: Load previously calculated solution into workspace.

AnsNumVars ← Number of different answers in the solution
AnsMatrix ← Matrix containing all of the *AnsNumVars* answers for the exercises
AnsDim ← Matrix containing the dimensions of each of the answers in the *AnsMatrix*

comment: Compare all student-generated variables with solution answers, recording the number of correct answers in *numcorr*.

numcorr ← 0
for *i* ← **to** *AnsNumVars*

do
- *AnsCurrent* ← Portion of *AnsMatrix* defined by i^{th} row of *AnsDim*
- **for** *j* ← 1 **to** *NumVars*
- **do**
  - *NameCurrent* ← j^{th} variable name in *VarNames*
  - **if** The variable in *NameCurrent* and the *AnsCurrent* variable are the same matrix size
  - **then**
    - **if** |*NameCurrent* variable – *AnsCurrent*| < *eps* for all matrix elements
    - **then** *numcorr* ← *numcorr* + 1

comment: If there is a set of student-generated variables in the workspace that is correct, return yes, else return no.

if *numcorr* = *AnsNumVars*
then *correct* ← *yes*
else *correct* ← *no*

The function is executed with no arguments and a binary result (*yes* or *no*) is returned in the variable *correct*, indicating whether the student solution is correct or incorrect.

EvalText is executed after the student, using an Exercise window interface such as the one described in Fig. 1, has solved a MATLAB exercise, producing a set of student variables in the workspace. The function compiles the names of each of the student-generated variables and places this list in the variable *VarNames* and the number of names in this list is placed in *NumVars*. The previously calculated solution is then loaded into the workspace. This solution could consist of a number of matrices of arbitrary size. Since all variables are compared, knowledge of the number of solution variables, the matrix size of each variable and the solution content is required. This is implemented using three solution variables: *AnsNumVars*, a scalar, which contains the number of answers in the solution; *AnsMatrix*, a matrix, which contains all of the answers; and *AnsDim*, a matrix, which gives the matrix dimensions of each of the answers present in *AnsMatrix*.

The comparison of the student solution to the actual solution is performed using the two **for** loops in Algorithm 1. The *AnsNumVars* answers in the precalculated solution are compared with the *NumVars* variables in the student solution. The number of matching comparisons are recorded in *numcorr*. Note that *eps* is a MATLAB variable defining the floating point relative accuracy and it is used in the comparison to handle possible truncation error.

Finally, if *numcorr* is equal to the number of answers, *AnsNumVars*, then the student solution is correct and this is indicated in the return variable, *correct*. The *EvalText* function is implemented as a MATLAB script that is placed in the file EvalText.m. The code for this MATLAB script is given in the Appendix.

MATLAB engine interface

The variable comparison function *EvalText*, described in the previous subsection, is a MATLAB script that is executed using a custom DLL routine in the MATLAB Engine Interface, as described in Fig. 2. This DLL routine is called *ExNoResult* and a pseudocode representation of this routine is given in Algorithm 2.

Algorithm 2 ExNoResult*(MatlabCommand)*

comment: This function is used to execute MATLAB commands that do not return a value.

comment: *engOpen* is a MATLAB DLL function that starts the MATLAB engine.

ep ← *engOpen(NULL)*

if *ep* does not contain a unique engine identifier
then Display the message,
'Error starting MATLAB engine!'

comment: *engEvalString* is a MATLAB DLL function that executes the *MatlabCommand* using the MATLAB engine specified by the *ep* identifier.

eval ← *engEvalString(ep,MatlabCommand)*

if *eval* indicates that the MATLAB engine is not running
  then Display the message, 'Error evaluating the command!'

This routine can be used to execute any MATLAB command, though it will not return any results, as the name implies. For example, if the student solution is being checked, the Computer-based System Program (CSP) executes *ExNoResult(IsCorrect = EvalText')*. The first step in Algorithm 2 is to start MATLAB using the *engOpen* routine. This is an Application Programming Interface (API) routine supplied with MATLAB [13]. Once MATLAB has been started, the MATLAB command *IsCorrect = EvalText* is executed using *engEvalString*, which is another API routine supplied with MATLAB. This will use the MATLAB engine to execute *EvalText* and place the result in the variable *IsCorrect*. The CSP must now retrieve the IsCorrect result from the MATLAB workspace to determine if the student solution is correct. The result can be retrieved using another DLL routine called *GetResult*, and a pseudocode representation of this routine is given in Algorithm 3.

Algorithm 3 GetResult(*VariableName*)

comment: This function is used to retrieve the contents of a variable in the MATLAB workspace.

comment: *engOpen* is a MATLAB DLL function that starts the MATLAB engine.

ep ← *engOpen(NULL)*

if *ep* does not contain a unique engine identifier
  then Display the message, 'Error starting MATLAB engine!'

comment: *engGetArray* is a MATLAB DLL function that copies the contents of the workspace variable *VariableName* to a mxArray structure and returns the address location of this structure. *mxGetPr* returns the contents of the newly created mxArray as a real number, which is then converted to an integer and placed in the return variable *answer*.

answer_location ← *engGetArray(ep, VariableName)*
real_answer ← *mxGetPr(answer_location)*
answer ← *real_answer* converted to an integer

if *answer* is not a binary value, 1 or 0
  then Display the message, 'Error getting result!'

return(answer)

For example, *GetResult('IsCorrect')* will retrieve the contents of the *IsCorrect* variable from the MATLAB workspace. This information can then be used by the CSP to inform the students if their solution is correct or incorrect.

One more DLL routine is required to execute MATLAB commands entered in the Student Exercise window described in Fig. 1. This DLL routine, called *ExecCmd*, executes the student-entered command and places the response produced by MATLAB in a file. The contents of this file are then displayed in the Student Exercise window by the CSP. A pseudocode representation of the *ExecCmd* routine is given in Algorithm 4.

Algorithm 4 ExecCmd(*MatlabCommand, FileName*)

comment: This function is used to execute the command, *MatlabCommand*, and place the result in the file *FileName*.

Allocate *BUFFER_SIZE* memory locations for variable *buffer*.
Fill the *BUFFER_SIZE* memory locations of *buffer* with null characters.

comment: *engOpen* is a MATLAB DLL function that starts the MATLAB engine.

ep ← *engOpen(NULL)*

if *ep* does not contain a unique engine identifier
  then Display the message, 'Error starting MATLAB engine!'

comment: *engOutputBuffer* is a MATLAB DLL function that specifies a memory buffer, *buffer*, of size *BUFFER_SIZE*, for the MATLAB engine specified by the *ep* identifier. *engEvalString* is a MATLAB DLL function that executes the *MatlabCommand* using the MATLAB engine specified by the *ep* identifier.

engOutputBuffer(ep,buffer,BUFFER_SIZE)
eval ← *engEvalString(ep,MatlabCommand)*

if *eval* indicates that the MATLAB engine is not running
  then Display the message, 'Error evaluating the command!'

if there is something in *buffer*
  then Write the contents of *buffer* to the file *FileName*

For example, if the student entered the MATLAB command, *whos*, in the Student Exercise window, the CSP would execute *ExecCmd ('whos','OutFile')*. This would execute *whos* and the list of workspace variables would be written to the file *OutFile*. The CSP would then display the contents of this file in the Exercise window.

IMPLEMENTATION EXAMPLE

The self-checking interface described in this paper has been implemented in the computer-based tutorial, *M-Tutor: An Introduction to MATLAB* [8]. An exercise from this tutorial is described here to demonstrate how this self-checking interface functions. The Student Exercise window described in Fig. 1 has been implemented as a specially designed Exercise window, which is the physical interface for the student to solve the MATLAB-based exercises. An example of this Exercise window, showing an exercise to be solved, is given in Fig. 3. The exercise problem to be solved is displayed in the top left-hand sub-window. The user can select the Hints button, which displays a list of hints to aid the student in solving the problem. The student solution to the problem is entered in the center subwindow, which is labeled Enter MATLAB Commands Here. This window has close to the full functionality of the MATLAB Command window, including previous command access using the arrow keys. The student enters a MATLAB command and hits the Return key and the result is displayed in the MATLAB Response subwindow. This response is the exact response you would see if the command were executed in the actual MATLAB Command window. This is implemented, as described in the section 'MATLAB Engine Interface', by evaluating the commands using the MATLAB engine through a DLL interface. Once the solution has been entered, the student selects the Evaluate button and a correct or incorrect indication is displayed. The evaluation method uses the MATLAB engine to compare the workspace contents and object variables of a correct solution with the student solution. The student can access an example of a correct solution from the Hints menu after using the Evaluate button.

A detailed description of the process used to initialize, enter and evaluate the student solution is given in the following section.

Student solution

The student opens the Exercise window after covering the associated material in the tutorial. Prior to opening the Exercise window, the CSP uses a DLL routine from the MATLAB Engine Interface to initialize the workspace. For the exercise in Fig. 3, the following command is executed: *ExNoResult('X=pascal(3);')*. This command generates a Pascal matrix of order 3. The variable, *X*, is now in the MATLAB workspace ready for the student to use to produce the solution to the exercise problem in Fig. 3.

Solving the problem involves entering the appropriate commands in the Exercise window.

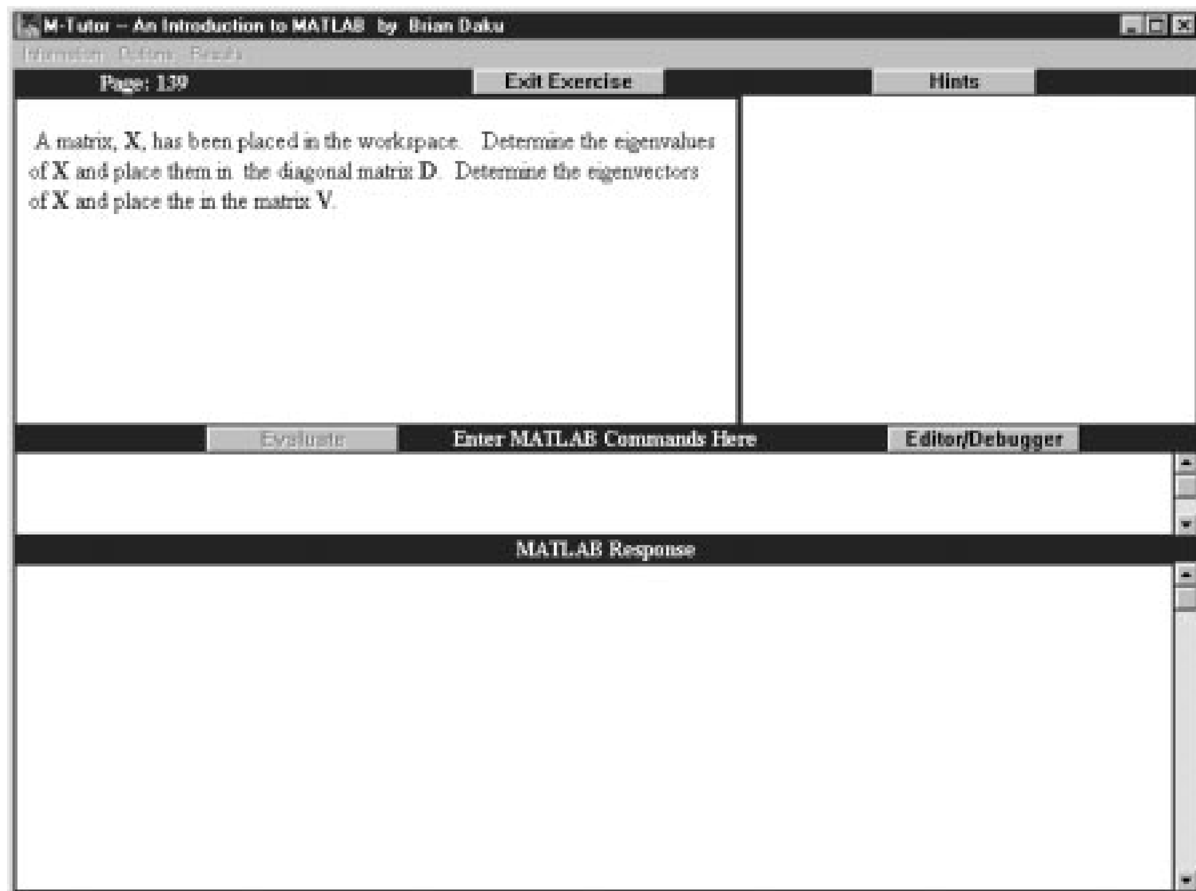

Fig. 3. Exercise window from the MATLAB tutorial.

In this case the variables V and D can be generated by entering the command, *[V,D] = eig(X)*, and executing the command by hitting Enter or Return on the keyboard. The CSP takes the entered MATLAB command and produces the DLL command, *execcmd('[V,D] = eig(X)','reply.txt')*, to execute the command in the MATLAB engine and place the resulting response in the file *reply.txt*, which is then displayed in the Exercise window. The result of this operation is displayed in the Exercise window of Fig. 4. This figure also shows the solution in the upper right subwindow, below the Hints menu. This solution is only accessible to the student after the student selects the Evaluate button, which performs the evaluation procedure that is described in the next section.

Exercise evaluation

The procedure to determine whether the student solution is correct is started when the Evaluate button in Fig. 4 is selected. The MATLAB function *EvalText*, described in the section 'Self-Checking Module', is executed using the DLL command *ExNoResult('IsCorrect = EvalText')*. This command uses the MATLAB engine to execute *EvalText*, placing the result in the workspace variable *IsCorrect*. Reviewing Algorithm 1, the first step is to enumerate the student Solution Variables. This will result in the following variables being placed in the workspace:

$$VarNames = \begin{bmatrix} \text{'D'} \\ \text{'V'} \end{bmatrix}$$

$$NumVars = 2$$

The solution variables, which are in the file filename.mat, are loaded into the workspace using the DLL function *ExNoResult*. The solution variables and their content are:

$$AnsNumVars = 2$$

$$AnsDim = \begin{bmatrix} 1 & 3 & 1 & 3 \\ 4 & 6 & 1 & 3 \end{bmatrix}$$

$$AnsMatrix = \begin{bmatrix} 0.5438 & -0.8165 & 0.1938 \\ -0.7812 & -0.4082 & 0.4722 \\ 0.3065 & 0.4082 & 0.8599 \\ 0.1270 & 0 & 0 \\ 0 & 1.0000 & 0 \\ 0 & 0 & 7.8730 \end{bmatrix}$$

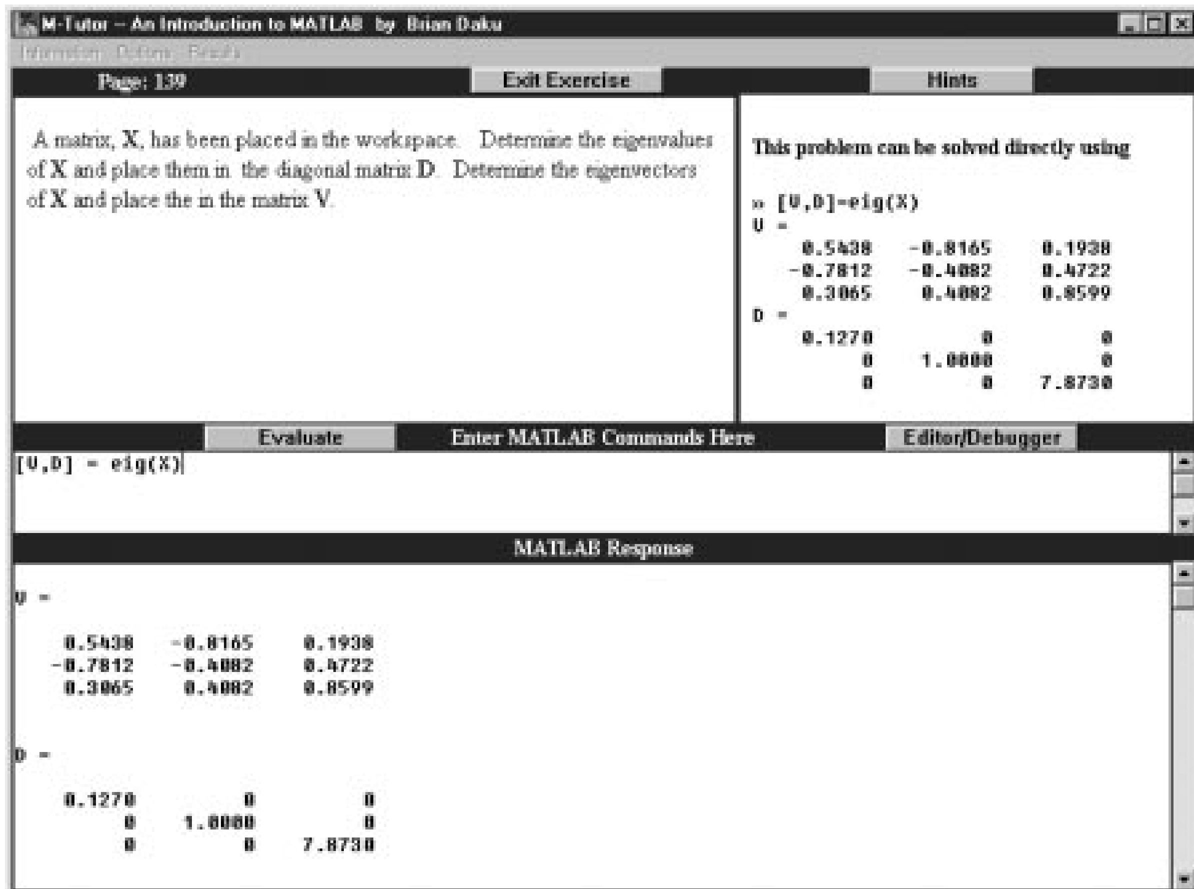

Fig. 4. Exercise window with the exercise completed. The answer is shown below the Hints menu. The answer can be displayed using the Hints menu once the Evaluate button has been selected.

EvalText compares each of the solution variables with each of the student-generated variables in the workspace. The contents of all of the solution variables are stored in a single matrix, *AnsMatrix*, and these are individually accessed using the indices that are stored as rows in the *AnsDim* variable. For example, for $i = 1$, $AnsCurrent = AnsMatrix(1:3, 1:3)$, which is the matrix:

$$AnsMatrix(1:3, 1:3) = \begin{bmatrix} 0.5438 & -0.8165 & 0.1938 \\ -0.7812 & -0.4082 & 0.4722 \\ 0.3065 & 0.4082 & 0.8599 \end{bmatrix}$$

Basically, *EvalText* compares the contents of the student variable D with the matrix $AnsMatrix(1:3, 1:3)$ and with $AnsMatrix(4:6, 1:3)$; if there is a match, *numcorr* is incremented by one. Similarly, the content of V is compared with the same two matrices in *AnsMatrix*, and *numcorr* is incremented if there is a match. Finally, if the content of *numcorr* is equal to *AnsNumVars*, the variable *correct* is set and the function completes returning the value of *correct*.

SUMMARY

This paper describes the construction of a self-checking interface that can be used to solve and evaluate exercises in computer-based educational tools. A unique feature of this system is that it uses MATLAB both to evaluate the student-generated solution to the exercises and provide feedback to the student. This self-checking interface has been implemented in a computer-based tutorial that is used to learn MATLAB and an example exercise has been taken from this tutorial to demonstrate how this interface works.

The main advantage of this self-checking interface is that it is a powerful method for effectively reinforcing material covered in computer-based tutorials. This was demonstrated in a recent (January 2001) evaluation of the M-Tutor tutorial by second year Electrical Engineering students. A group of 79 students were required to learn MATLAB on their own, though they did not have to use the M-Tutor tutorial. A further 71 students used the tutorial and 52 of these did the exercises, of which 42 felt the exercise interface was very useful for reinforcing the MATLAB concepts. Though these results are subjective, it does imply that the self-checking interface is a useful approach that should be pursued.

APPENDIX

The following is the MATLAB code that is used to implement the *EvalText* function. This code is placed in the file EvalText.m.

```
function correct=EvalText

%EVALTEXT

% x=EvalText
%
%This function is used to evaluate the
%exercises that produce
%alphanumerical text-based solutions.
%The programmer must ensure that the
%appropriate solution
%variables to the exercises have been
%generated and saved in
%'filename.mat', as described in the
%programmers' documentation.
%The value returned is a 1 if correct and
%a 0 if incorrect.

VarNames=evalin('base','who');
NumVars=length(VarNames);
correct=0;
load filename
numcorr=0;
for i=1:AnsNumVars
 AnsCurrent=AnsMatrix(AnsDim(i,1):
  AnsDim(i,2),AnsDim(i,3):
  AnsDim(i,4));
 for j=1:NumVars
  NameCurrent=char(VarNames(j));
  if all(size(evalin('base',
   NameCurrent))==size
   (AnsCurrent))
   if all(all(abs(evalin('base',
    NameCurrent)-AnsCurrent)<eps))
    numcorr=numcorr+1;
    break
   end
  end
 end
end
if numcorr==AnsNumVars
 correct=1;
end
```

Acknowledgements—The author would like to thank the Peter N. Nikiforuk Innovative Teaching and Learning Centre at the University of Saskatchewan and the Canadian Industrial Research Assistance Program (IRAP) for financial support in the development of the M-Tutor tutorial. The author would like to acknowledge Karl Lehmann, who implemented most of the software, and Ron Bolton for help in implementing the DLLs.

REFERENCES

1. R. Shiavi, Learning signal processing using interactive notebooks, *IEEE Transactions on Education*, **42**(4) (1999).
2. S. E. Fisher and E. Michielssen, Mathematica assisted web-based antenna education, *IEEE Transactions on Education*, **41**(4) (1998).

3. S. Pomeranz, Using a computer algebra system to teach the finite element method*, The International Journal of Engineering Education*, **16**(4) (2000) pp. 362–368.
4. H. S. Hinton et al., A technology-enhanced learning environment for a graduate/undergraduate course on optical fiber communications, *Frontiers in Education 2000* (2000) pp. F1E-1 to F1E-6.
5. T. W. Ng, Creating a multiple-choice self-marking engine on the internet, *International Journal of Engineering Education*, **16**(1) (2000) pp. 50–55.
6. MathWorks Inc., Mathworks website for MATLAB, http://www.mathworks.com
7. MathWorks Inc., MATLAB Based Books for use with MATLAB, Simulink, Toolboxes, and Blocksets, http://www.mathworks.com/books
8. B. L. F. Daku, *M-Tutor: An Introduction to MATLAB*. Prentice-Hall, Canada, ISBN 0-13-083396-7, CD-ROM (1999).
9. B. L. F. Daku and K. D. Jeffrey, Development of an interactive cd-rom-based tutorial for teaching MATLAB, *IEEE Transactions on Education*, **44**(2) (2001).
10. B. L. F. Daku and K. D. Jeffrey, An interactive computer-based tutorial for MATLAB, *Frontiers in Education 2000* (2000) pp. F2D-2 to F2D-7.
11. B. L. F. Daku, M-Tutor website, http://www.m-tutor.usask.ca
12. J. Siglar, Multimedia Authoring FAQ, http://www.tiac.net/users/jasiglar/faz-index.html
13. MathWorks. *MATLAB Application Program Interface Guide*, MathWorks Inc. (January 1998).

Brian Daku received his B.A.Sc degree in Electrical Engineering from the University of Waterloo, Waterloo, Ontario, in 1980 and his M.Sc. and Ph.D. degrees in Electrical Engineering from the University of Saskatchewan, Saskatoon, Saskatchewan, in 1987 and 1990 respectively. He worked for SED Systems Inc., Saskatoon, from 1980 to 1984, where he was involved with microprocessor applications in satellite receivers and industrial products. In 1986/87 he was a Visiting Research Scholar at the University of Edinburgh, Scotland. He joined the University of Saskatchewan's Department of Electrical Engineering in 1990. His research interests are in microseismic fault localization, wireless communication systems, and digital signal processing.