

Building and Using Interactive Mathematics Software
Robert Decker
University of Hartford

Computer algebra systems are not strong at “fast feedback” exploration. In order to see how a parameter change affects the graph of a function or differential equation one must edit and then reissue a command, which is slow and inconvenient.

Tcl/Tk is a free platform independent (mac, windows, linux, unix) language that can be used to quickly build interactive mathematics programs. I have created a number of helper functions (components), which make it even easier for a mathematics educator to create little programs to illustrate particular concepts. These interactive programs can be used as stand-alone programs or as front-ends to computer algebra systems (explore and then transfer the work to the computer algebra system).

I will discuss some interactive graphing programs that illustrate parameter changes for functions and differential equations, and changes in initial conditions for differential equations (it becomes very easy to generate a complete phase portrait when done interactively). Such investigations are useful, in that the dynamic exploration of an equation can lead much more quickly to parameter values of interest than is possible with other types of software. Once the interesting region of parameter values is located, the work can be exported, and a computer algebra system can be brought in for further investigation and to produce high quality output and reports.

I will also discuss benefits to both teachers and students and some possible research directions in mathematics education related interactive software, as well as to this “write and then use” approach.

Some History

Before the advent of windows-based computers and the common use of graphing calculators and computer algebra systems, a number of mathematics educators used programming languages such as True Basic or Pascal to create simple programs that students could use to explore mathematical concepts visually. Many of those who used to dabble in writing their own programs have moved to using either graphing calculators or computer algebra systems such as Maple or Mathematica, all of which are programmable.

Also, it was much easier to write a simple program to graph functions, area under a curve, tangent lines and so on for a DOS based computer than it is now for any of the graphically based operating systems. Anyone not well versed in the current object-oriented method of programming, which usually involves extensive knowledge of C++ or Java, will find it time consuming to write even simple programs for any of the common operating systems.

Graphing calculators suffer primarily from lack of speed and limited screen resolution. While one can write a program to interactively change a parameter to see its effect on one or several graphs, the slow speed of graphing lessens the visual impact on the student. The almost immediate response that one obtains from a computer makes a big difference in how students come to understand the effect of a parameter change.

Computer algebra systems overcome one of the limitations of calculators, that of speed. They don't, however, overcome the second limitation mentioned above, in that they don't operate

interactively in the following sense. The current round of computer algebra systems (I will focus on Maple and Mathematica) operate on a command-based principle. A command is typed in on a command line, the command is then executed, and then the graph is produced. The exception to this when an animation is created; in this case an interactive slider is created allowing the user to switch between the different frames of the animation. To explore parameter values in a range not contained in the original animation a new animation must be created, a time-consuming process.

Finally, one of the objectives of some mathematics educators in writing their own programs was to allow and/or require their students to alter the program in order to extend its capabilities. Thus the student would learn a bit about programming, certainly a useful skill in today's world (though the emphasis of the process would not be on programming per se).

Back to Programming

Enter Tcl/Tk, a platform independent, interpreted, programming language, which can be used to create simple programs with a modern graphical user interface. The language actually consists of two pieces; the Tcl component is a purely character-based programming language, which is especially good at string manipulation. The Tk component is used to create the graphical user interface (both Tcl and Tk were created by John Ousterhout). The tools required to create Tcl/Tk programs are available at no cost, and the programs created can be freely distributed. Such programs will run on Window/Macintosh/Linux/Unix computers with no alteration. The learning curve for Tcl/Tk is quite short, especially if one is familiar with another language such as Basic or Pascal. It is especially easy to create the components of a graphical user interface, such as multiple windows, entry boxes, scroll bars, sliders and menus. Thus one can create, for example, a slider which when moved instantly reflects the effect of a parameter change on the graph of a function. Finally, and perhaps most importantly, because Tcl/Tk is an interpreted language, it easily handles mathematical expressions without the need for the programmer to write a mathematical parser (the capability is built in to the interpreter).

For those who need or want the full capabilities of a computer algebra system, these little interactive programs can be quite useful. I have written a converter which outputs Maple commands so that work done in a Tcl/Tk program can be transferred to Maple (I would be happy to write converters for other computer algebra systems if there is interest). This overcomes the main limitation of computer algebra systems, the lack of interactivity. The role of the computer algebra system is enhanced, not eliminated; one can more easily find parameter values or initial conditions that lead to interesting behavior in a function or differential equation, then transfer the work to a computer algebra system. Once back in the computer algebra system, the commands can be tweaked to get the final desired result, and text can be added to create a report.

Before getting too excited about this language, let's get to the drawbacks. Because Tcl/Tk is an interpreted language, it is slower than compiled languages. Thus most experts do not recommend that applications that require intensive number crunching be written in this language. It is possible to combine Tcl/Tk with C or C++ so that the graphical interface is handled in Tcl/Tk and the number crunching is done in a C or C++ subroutine, but this detracts from the simplicity of the process of writing programs.

The speed of the current round of computers has helped to overcome the slowness of the language. I have found that for a program that graphs a small number of functions or differential

equations, the change of the value of a parameter by using a graphical slider is reflected nearly instantly in the graph for computers with a 700-megahertz processor. For computers with a processor as slow as 100 megahertz, the programs are still usable, but the changes in the graph appear jerky.

Getting Started with Tcl/Tk

Since Tcl/Tk is an interpreted language, you first need to download and install the free Tcl/Tk interpreter program, called Wish (see my website uhaweb.hartford.edu/rdecker, or go to sourceforge.net/projects/tcl/ for the latest release). There is a version of the Wish program for all of the supported platforms; once this program is installed, you can run any Tcl/Tk program (programs are also called scripts). Once Wish is started up, you can type in commands interactively, or run a script contained in a separate file. Use the Source option on the File menu to choose the Tcl/Tk script that you want to run, or type “source filename.tcl” at the prompt to run the script contained in filename.tcl. On a Windows computer you can also just double click any file with a .tcl extension to run the program in that file.

I do not want to try to teach the reader Tcl/Tk; there are some books available about the language (see [3], [4], [5], [6]), as well as a great deal of information on the World Wide Web. Try the following:

- Tcl Developer Site, www.tcl.tk
- The Tccler's Wiki, mini.net/tcl/
- comp.lang.tcl (usenet newsgroup)

I will, however, present a few of the features of the language through some examples.

A Simple Example of Tcl/Tk at Work

In order to show the simplicity of the language, and how easily it can be used to create graphical user interfaces, I will present the code contained in a file on my website called `simple_grapher.tcl`, which also illustrates many of the techniques I use to build more complex graphing programs.

The program that follows the screen shot below produces a simple function-graphing program with a slider to adjust the parameter “a”; feedback is practically instantaneous. Any standard math function can be entered and will be graphed on the region $-10 \leq x \leq 10$, $-10 \leq y \leq 10$. You must use \$x for the variable and \$a for the parameter (use of the dollar sign is explained later). The program produces a window on your computer as shown in Figure 1 (the program looks similar on all supported operating systems; the one below is taken from Windows). If you type in these statements yourself, save them in a file called `simple_grapher.tcl`.

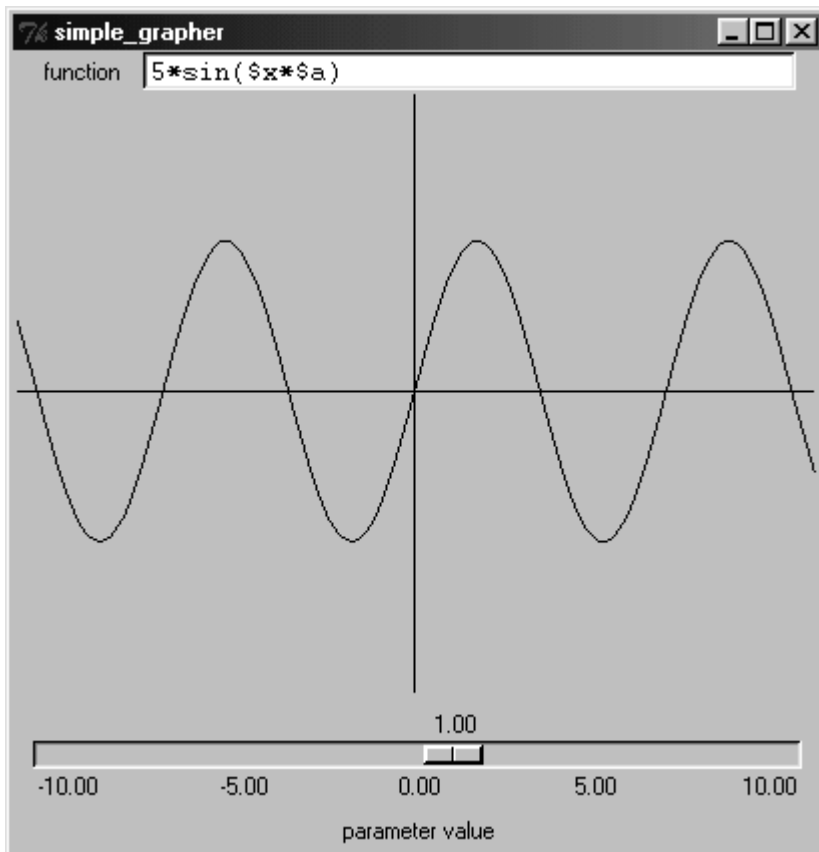


Figure 1

```

# main program
set fontname ansifixed                                ;# use fixed width font for
                                                       ;# math expressions

set expr1 {5*sin($x*$a)}                             ;# the function to be graphed

set xmax 10.0                                        ;# these statements set properties of
set xmin -10.0                                       ;# the graph
set ymax 10.0
set ymin -10.0
set xpicmax 400
set ypicmax 300
set numpoints [expr $xpicmax/2]
set xinc [expr ($xmax-$xmin)/$numpoints]

set ax [expr $xpicmax/($xmax-$xmin)]                ;# these statements set the constants
set ay [expr -$ypicmax/($ymax-$ymin)]                ;# that are used to convert between
set bx [expr -$xpicmax*$xmin/($xmax-$xmin)]          ;# pixels and x-y coordinates
set by [expr $ypicmax*$ymin/($ymax-$ymin)+$ypicmax]

set amin -10                                         ;# these statements set properties of

```

```

set amax 10                                ;# the slider used to control the
set ares [expr ($amax-$amin)/1000.0]      ;# parameter "a"
set atic [expr ($amax-$amin)/4.0]

set a 1                                    ;# initializes the parameter "a"

```

Anything following the # sign on a line is a comment. Statements can end with either a semicolon or no symbol at all, but if a comment follows the statement the semicolon is needed. The set command is used to assign a value to a variable. A variable can be assigned any string or numerical value (similar to computer algebra systems). One feature of the language that is different than most computer languages is that once a variable is assigned a value, a dollar sign must be placed in front of the variable to access that value. Thus a variable such as myVariable simply represents itself (the string myVariable), but \$myVariable represents the value stored in that variable. Also, rather than using the ^ symbol for exponentiation, in Tcl/Tk the "pow" function is used as in the C language (instead of 2^3 use pow(2,3)).

Brackets are used to enclose commands, which can be thought of as functions or procedures (either predefined or user defined). In order to do arithmetic, the expr command must be used (again, this is different than in most languages). The above code defines several variables, which will be used in the program.

The next piece of code creates part of the graphical user interface (GUI).

```

#function frame
frame .f1                                  ;# container widget
pack .f1
label .f1.l1 -text "function "            ;# label widget, child of .f1
pack .f1.l1 -side left
entry .f1.e1 -textvariable expr1 -width 40 -font $fontname ;# entry widget, child of .f1
pack .f1.e1 -side left
bind .f1.e1 <Return> {                    ;# binding, so that when the
    .c delete all                          ;# function is changed and
    DrawAxes $xmin $xmax $ymin $ymax $ax $ay $bx $by ;# Return pressed, the graph
    Plot $xmin $ax $ay $bx $by $xinc $expr1 $numpoints ;# is redrawn
}

```

The components of a GUI are called widgets in Tcl/Tk (we are now using the Tk part of the language). First we create a frame widget (an empty container), which will contain a label widget with the word "function", and an entry widget, which contains the function formula. Entry widgets are used extensively in my approach to building mathematics software; all of the properties of a graph can be adjusted through the entry widgets. After a widget is created, it must be "packed" for it to appear on the screen; packing establishes where the widget will appear. This frame and the widgets it contains appear at the top of the window shown above.

Finally, we establish a "binding", which is a rule that associates a widget, an event, and an action. In this case, when the Return key is pressed (event) while in the entry widget, the graph is erased, and then the axes and function are redrawn. The widget .c is a canvas widget used for making drawings, and DrawAxes and Plot are user defined commands (procedures)

which I have created; the arguments of the function follow the function name separated by spaces.

Next the graph region (canvas widget) is created.

```
#graph region
canvas .c -width $xpixmap -height $ypixmap
pack .c
```

Its width and height are given in pixels, stored in the variables `xpixmap` and `ypixmap`.

The final part of the main program creates the slider (scale widget) and a label with the words “parameter value.”

```
#parameter scale widget
scale .s1 -orient h -tickinterval $atic -length 4i -variable a \ ;# create slider
    -from $amin -to $amax -resolution $ares \ ;# \ used for line continuation
    -width 10 -command { ;# the -command option is
.c delete all ;# equivalent to adding a
    DrawAxes $xmin $xmax $ymin $ymax $ax $ay $bx $by ;# binding
    Plot $xmin $ax $ay $bx $by $xinc $expr1 $numpoints
    set null}
pack .s1
label .l2 -text "parameter value"
pack .l2
```

The `-variable` option establishes an automatic link between the value stored in the variable “a” and the numerical value of the slider. Instead of adding a binding to the scale widget, it has a built-in binding resulting from the `-command` option. Tcl/Tk appends the numerical value of the slider to the end of the command script (right after “null” in this case), but we do not need that value since it is already tied to “a” with the `-variable` option, hence we store the value in the variable “null” which is not used.

The entire main program has now been presented. The rest of the program consists of two procedures, `Plot` and `DrawAxes`. Arguments of procedures appear in the first set of braces, and the statements appear in the second set of braces. Note: the procedures must appear before the main program in the file `simple_grapher.tcl`.

```
#a subroutine which draws the graph of the function
proc Plot {xmin ax ay bx by xinc expr1 numpoints} { ;# procedure name and incoming
    ;# arguments
    global a ;# the parameter controlled by
    ;# the slider
    set x $xmin ;# first x-coordinate plotted
    set y [expr $expr1] ;# evaluates function to get y
    set xp [expr {round($ax*$x+$bx)}] ;# convert from x-y coordinates
    set yp [expr {round($ay*$y+$by)}] ;# to pixels
    set coords [list $xp $yp] ;# store pixel coordinates in a list
```

```

for {set index 1} {$index<=$numpoints} {incr index} {
  set x [expr {$x+$xinc}]           ;# loop which evaluates function
  set y [expr $expr1]               ;# for all x-coordinates, converts
  set xpic [expr {round($ax*$x+$bx)}] ;# to pixels and adds coordinates
  set ypic [expr {round($ay*$y+$by)}] ;# to the list of coordinates
  set coords [concat $coords $xpic $ypic]
}
.c create line $coords               ;# draws line segments connecting
}                                   ;# all of the points in the list called
                                   ;# coords

#a subroutine that draws the coordinate axes
proc DrawAxes {xmin xmax ymin ymax ax ay bx by} {

.c create line [expr {$xmin*$ax+$bx}] [expr $by] \      ;# draws x-axis in red
               [expr {$xmax*$ax+$bx}] [expr $by] -fill red
.c create line [expr $bx] [expr {$ymin*$ay+$by}] \      ;# draws y-axis in red
               [expr $bx] [expr {$ymax*$ay+$by}] -fill red
}

```

As indicated in the comments, the first procedure simply loops through x -values, calculates the corresponding y -values (with the statement “set y [expr \$expr1]”), and then stores the x - y pairs in the list “coords.” The single statement “.c create line \$coords” draws all of the points in the list and connects them with line segments. The second procedure simply plots two lines, the x and y -axes.

Notice that to get values into a procedure you can either pass them in through the argument list, or declare them as global. With global variables, changes made in the procedure to the variable remain after the program leaves the procedure, whereas with variables passed in through the argument list, local changes are not reflected in the value of the variable being passed in (pass by value only is allowed). Finally, although nothing is returned with these procedures, you may return the value of a variable, including lists, so that any number of values may be returned (by putting them in a list).

More Complex Programs using Components

In order to move beyond the limitations of very simple programs such as the one discussed above, I have written a number of helper functions (components) that allow one to write more extensive programs. Programs written using these functions allow the user to resize the graph region, change the scale (zoom), combine different types of graphs (such as function graphs and differential equation graphs), change the names of the variables or display several coordinate systems at once with different graphs on each. Also, a function is provided which allows the user to enter formulas in “natural” mathematical form (no dollar signs are required, and the ^ symbol is used for exponentiation), and error trapping is employed to minimize program crashes.

To create the function grapher shown in Figure 2, with one slider and two functions, one would use the code below.

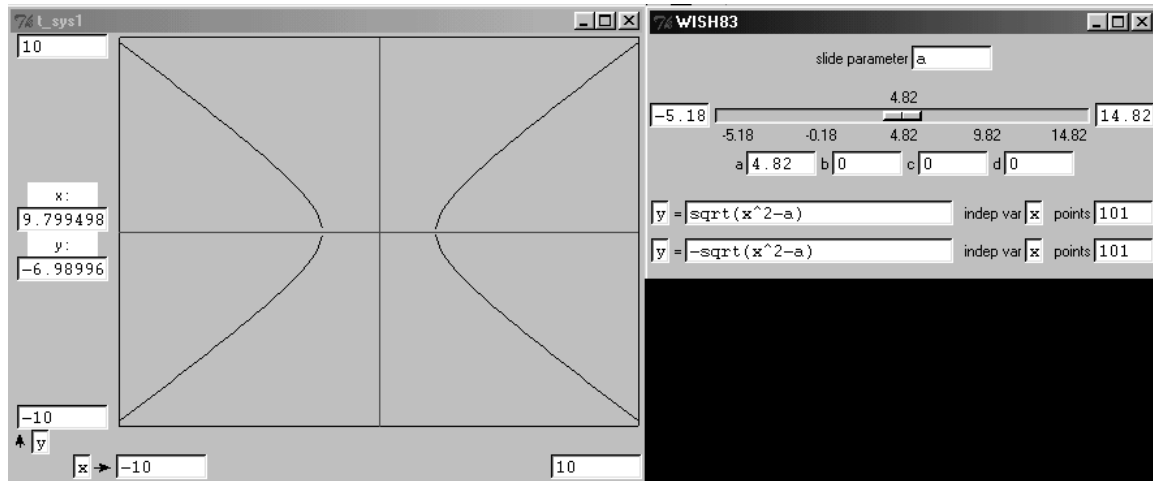


Figure 2

```
source modular1.tcl      ;# these commands add several other files to the program,
source slider.tcl       ;# much as "include" does in a C++ program
                        ;# in these files are initializations and all helper functions
```

```
set coordsysOut(sys1) [coordsys sys1 -10 10 -10 10 400 300 x y]
# creates a coordinate system with name sys1, from -10 to 10 in both
#directions, which is 400 pixels wide and 300 pixels tall, with variable names x and y
```

```
AddGraphDatabase {
    set plotCoordsysOut(sys1) [PlotCoordsys $coordsysOut(sys1)];
}
#plots the coordinate systems axes and bounding box, and saves the command
#in a global variable graphDatabase
```

```
# a simple function, the top of a hyperbola
set func(1) sqrt(x^2-a)      ;# define the function to be graphed
set indepVar(1) x           ;# define dependent and independent variables
set depVar(1) y
set numPts(1) 101          ;# number of points to plot
```

```
set funcFixedOut(1) [FixFunction $func(1) [list $indepVar(1) $depVar(1)]]
# converts the function in func(1) to Tcl/Tk format
```

```
AddGraphDatabase {
    set funcptsOut(1) [funcpts $hmin(sys1) $hmax(sys1) $numPts(1) $funcFixedOut(1) \
        $a $b $c $d];
    set plotFuncOut(1,sys1) [PlotFunc $funcptsOut(1) $coordsysOut(sys1)];
}
```

```
#creates the points to be plotted, picks a coordinate system to plot them on, and saves
#the command in the variable graphDatabase
```

```
NewFunction 1      ;# creates the GUI for this function
```

```
#a second function, the bottom of the same hyperbola
set func(2) -sqrt(x^2-a)      ;# see first function for explanations
set indepVar(2) x
set depVar(2) y
set numPts(2) 101
set funcFixedOut(2) [FixFunction $func(2) [list $indepVar(2) $depVar(2)]]
AddGraphDatabase {
  set funcptsOut(2) [funcpts $hmin(sys1) $hmax(sys1) $numPts(2) $funcFixedOut(2) \
    $a $b $c $d];
  set plotFuncOut(2,sys1) [PlotFunc $funcptsOut(2) $coordsysOut(sys1)];
}
NewFunction 2
```

The code here is actually shorter than that of the first program presented (my comments make it look longer than it is), but the program is considerably more sophisticated than the first one. The graph and function data appear in different windows so that you can arrange things the way you want on your screen. The graph window can be resized (to fill the entire screen if desired), and the letters used for the variables and their ranges can be adjusted right on the graph window. Also, when the cursor is over the graph window, the x and y coordinates of the cursor are displayed.

Up to four parameters can be used in either function (a, b, c, d), and the values of the parameters can be changed either through the entry boxes or by using the slider. The range of values for the slider can be adjusted, and I have employed a “snap back” slider, which always returns to the middle position, adjusting the minimum and maximum values accordingly (this allows one to use the slider to adjust a parameter to a value originally outside the range of the slider).

In the function window, the choice of letter for dependent and independent variable is up to the user (of course they must be the same as the variable letters in the plot window). If the roles of dependent and independent variable are switched in the plot window, the graph of the inverse of the function appears.

My last sample program, this one again created out of components, is a program that graphs a system of first order differential equations. Three coordinate systems are created, so that any of three views of the equation can be seen at once (phase plot with vector field, or either of the two possible time plots). Again, there is a slider, which can adjust any of four possible parameters (only two are used in this example). Also, I have included a binding which allows the user to set the initial conditions by clicking anywhere in the graph region; if the user drags the mouse while the button is clicked, one sees the solution curve change dynamically in all three views.

Finally, I have incorporated the ability to output Maple code. Just click the button labeled “Copy Maple Commands” and the appropriate Maple commands are copied to the clipboard; now switch to Maple and paste the commands at an input prompt. In Figures 3 and 4, I show screen shots of both the Tcl/Tk program and the output from Maple. The observant reader will notice that the equations correspond to those of a damped pendulum; being able to explain the relationship between these three graphs is something that every student of differential equations should be able to do.

Tcl/Tk Program

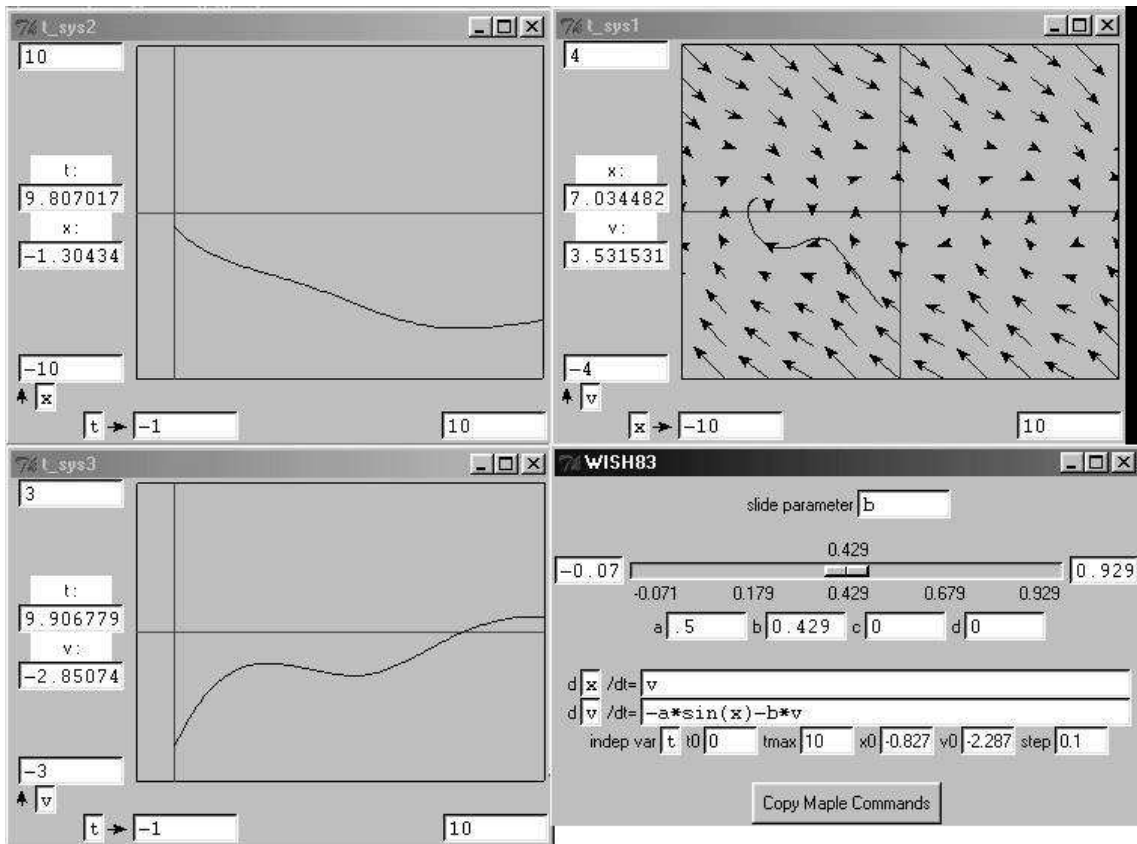


Figure 3

Maple Output

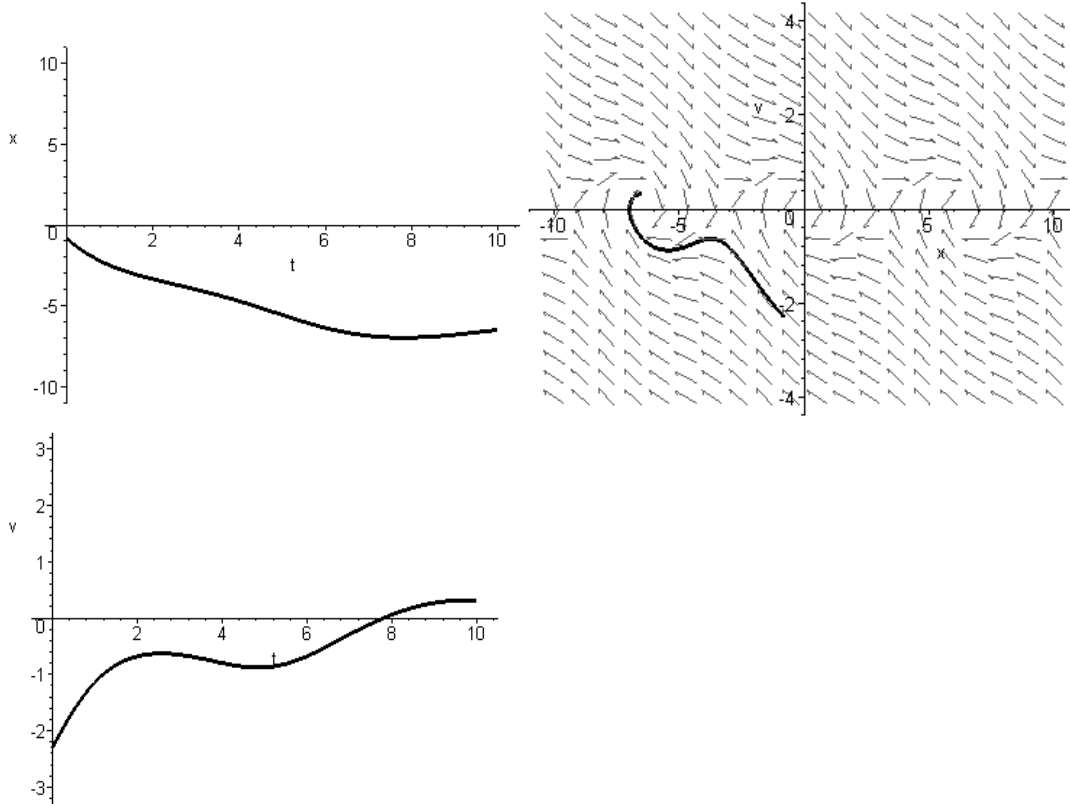


Figure 4

The code for this program is again quite short (see the previous program for comments which explain the statements, as there are many similarities).

```
source modular1.tcl
source slider.tcl

#set up three coordinate systems
set coordsysOut(sys1) [coordsys sys1 -10 10 -10 10 400 300 x y]
AddGraphDatabase {
    set plotCoordsysOut(sys1) [PlotCoordsys $coordsysOut(sys1)];
}
set coordsysOut(sys2) [coordsys sys2 -1 10 -3 3 \
    400 300 t x]
AddGraphDatabase {
    set plotCoordsysOut(sys2) [PlotCoordsys $coordsysOut(sys2)];
}
set coordsysOut(sys3) [coordsys sys3 -1 10 -3 3 \
    400 300 t y]
AddGraphDatabase {
    set plotCoordsysOut(sys3) [PlotCoordsys $coordsysOut(sys3)];
}
```

```

}
#define de system
set fDesys(1) y
set gDesys(1) -a*x
set indepVar(1) t
set fDepVar(1) x
set gDepVar(1) y
set indepMin(1) 0
set indepMax(1) 10
set initialDep1(1) 0
set initialDep2(1) 1
set step(1) 0.1
set solutionType(1) rk4sys

set fFixedOut(1) [FixFunction $fDesys(1) [list $indepVar(1) $fDepVar(1) $gDepVar(1)]]
set gFixedOut(1) [FixFunction $gDesys(1) [list $indepVar(1) $fDepVar(1) $gDepVar(1)]]

AddGraphDatabase {
  set desysOut(1) [$solutionType(1) $fFixedOut(1) $gFixedOut(1) $indepMin(1) $indepMax(1)\
    $initialDep1(1) $initialDep2(1) $step(1) $a $b $c $d];
  set plotFuncOut(1,sys1) [PlotFunc $desysOut(1) $coordsysOut(sys1)];
  set plotFuncOut(1,sys2) [PlotFunc $desysOut(1) $coordsysOut(sys2)];
  set plotFuncOut(1,sys3) [PlotFunc $desysOut(1) $coordsysOut(sys3)];
}
set desysList [concat 7 $desysList]
NewDesys 7

#vector field for same de system
set vectLen(1) 1 ;# initialize the maximum vector length
AddGraphDatabase {
  set deBasepts(1) [base_pts 11 $coordsysOut(sys1)]
  set deFieldpts(1) [field_pts $deBasepts(1) $fFixedOut(1) $gFixedOut(1) $vectLen(1)\
    $a $b $c $d];
  set plotFieldOut(1) [PlotField $deBasepts(1) $deFieldpts(1) $coordsysOut(sys1)];
}
# commands to give coordinates of vectors and plot them
ResetField 7 sys1 ;# resets max vector length to get a
eval $resetField ;# better looking vector field

```

Fourth order Runge-Kutta is used for plotting the differential equations, though either Euler or a standard second order method can be used also (all are included in the helper functions). It is a very powerful program, which can be used to investigate systems of differential equations much more quickly than is possible with a computer algebra system alone.

Pedagogical Considerations

Parameters play an important role in mathematics, but are often neglected in mathematics courses. The role of variable is stressed, but you will still find that in most mathematics texts, equations with both variables and parameters are somewhat rare. In calculus, for example, you will see many problems of the form $\int 3e^{2x} dx$, but few of the form $\int 3e^{ax} dx$. It is important to include more examples with meaningful parameters, and to get students to investigate the effect of parameter changes on the shape of a graph of a function or differential equation, so that they understand the roles of the parameters.

Over the last few years, I have spent a great deal of time having students go through this sort of exercise using graphing calculators and computer algebra systems (see [1] and [2]). The lack of interactivity, however, severely handicaps what a mathematics instructor can accomplish with students. When students see only discrete snapshots of functions as a parameter changes, and not the “in between” pictures, they sometimes don’t see what it is that the instructor wants them to see. Also, the tedious nature of editing a statement and reissuing the command detracts from the appeal of the process. Some fairly simple graphs can require several lengthy commands in a computer algebra system. For example, to generate graphic output for a report that is meant to show the effect of step size on the numerical solution to a differential equation would require at least six commands if three different views of the solution curve are desired; with a program I have created with the tools described here, the student can interactively find the parameter values that best illustrate the phenomenon under investigation, and then export the six Maple commands at once. Finally, it can be very difficult to pinpoint a parameter value where an important change occurs (such as a bifurcation in differential equations) using just trial and error and a computer algebra system (even for instructors).

I believe that some research is needed on how well students actually understand the role of parameters, and which teaching methods best increase this understanding. In particular, is it possible to produce a sufficient number of static “snapshots” to get across to a student the meaning of a continuous parameter, or is it necessary for most students to see the “in between” pictures and the real-time motion to achieve true understanding? Do students learn more by adjusting a parameter’s value using an interactive slider, or by actually retyping in various numbers? My own experience suggests some answers, but I believe that the mathematics community would benefit from some studies in this area.

Finally, I would like to suggest that having both students and teachers participate in the creation of these interactive programs has benefits. Both will be exposed to some computer programming, certainly a useful skill in today’s world. Also, having the tools to create sophisticated graphing programs without a lot of effort would encourage both students and teachers to get more involved in the creation of mathematics software, which can only help the rest of the mathematics community.

References

- [1] Calculus, Preliminary Edition, Robert Decker and Dale Varberg, Prentice Hall, 1993.
- [2] Bringing Calculus to Life, Robert Decker and John Williams, Prentice Hall, 1996.
- [3] Tcl and the Tk Toolkit, John K. Ousterhout, Addison-Wesley, May 1994.
- [4] Tcl/Tk in A Nutshell, Paul Raines, Jeff Tranter, Andy Oram (Editor), O'Reilly & Associates, May 15, 1999.
- [5] Tcl/Tk for Dummies, Tim Webster, IDG Books Worldwide, Inc., 1997.
- [6] Practical Programming in Tcl and Tk (3rd Edition), Brent B. Welch, Prentice Hall, Jan1, 2000.