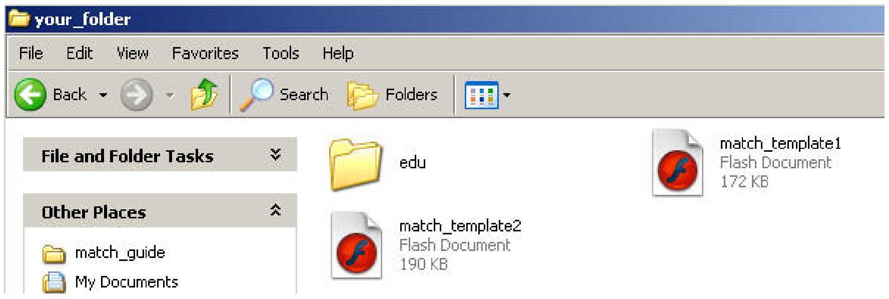
Flash Tools for Developers: Matching Formulas to Data
A Guide

This paper is a companion to the online article at the MathDL Digital Classroom Resources "Flash Tools for Developers: Matching Formulas to Data" by Doug Ensley and Barbara Kaskosz. This paper provides a description of the two easily customizable templates and the underlying ActionScript classes presented in the article. The complete source code for the templates and the classes discussed below can be downloaded from the article through the link match_formula.zip. In this article, we are assuming that your are familiar with the basics of Flash's authoring environment. If not, please download the PDF guide from our earlier article "Flash Tools for Developers: Function Grapher", or visit our MathDL Flash Forum Learning Center at

http://www.math.uri.edu/~flashcenter/

Introduction

Download match_formula.zip file and unzip it in a folder on your computer. You will see a match_formula folder which contains all the files related to the article. The ones you are particularly interested in are: the folder edu which contains all the necessary ActionScript classes (in a nested sequence of folders), and the two source files for the templates: match_template1.fla and match_template2.fla. Working right from the folder match_formula, you can open one of the two template fla files in Flash 8 Professional and begin exploring and customizing the template. Or, you can copy the edu folder and the templates into a new folder. For Flash to be able to find the classes, the folder edu must reside in the same folder as the template you are working on:

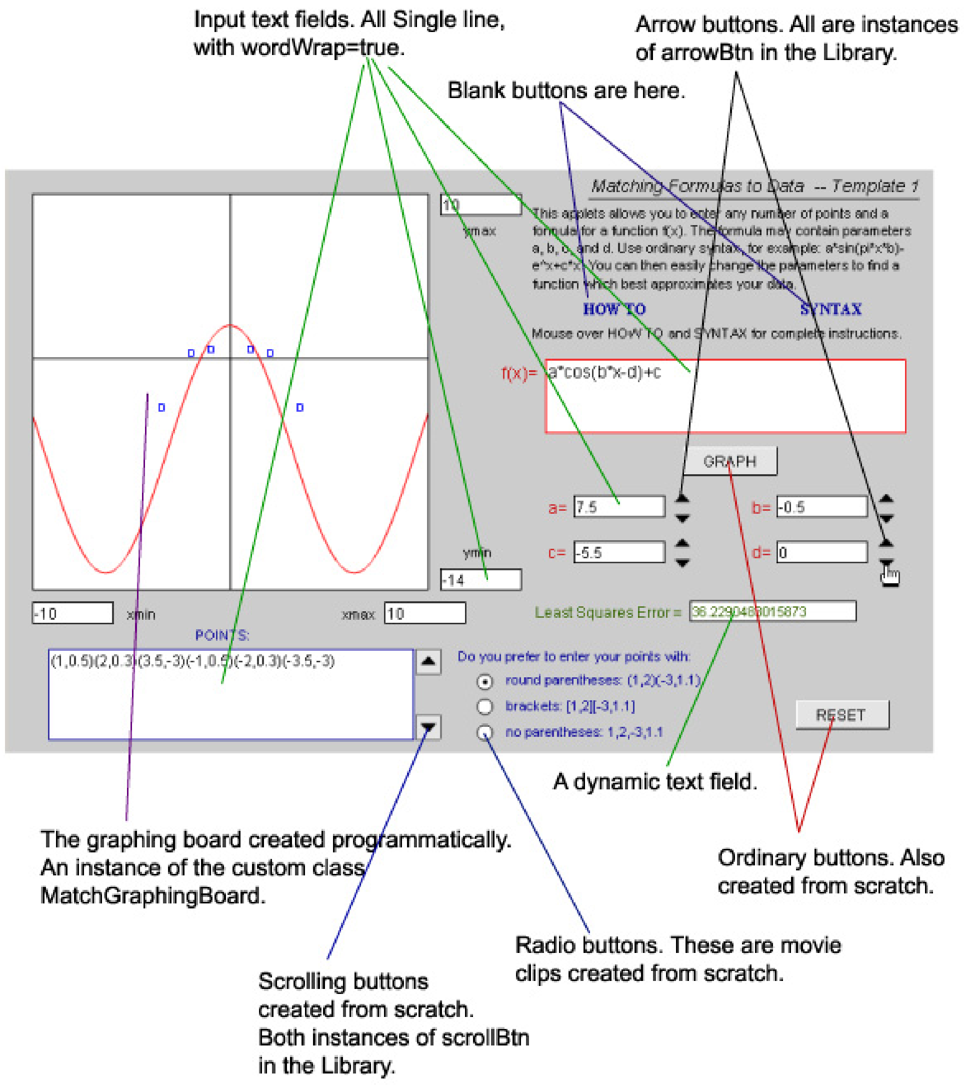


Picture 1

(The fla files are saved in Flash 8 format. If you work in Flash MX 2004, drop us an email. We will send you the fla files saved in Flash 2004 format.)

User Interface Elements

While in Flash, navigate to, and open one of the templates, say match_template1. Let's begin by testing the Movie. (Choose **Control** from the uppermost menu, and click **Test Movie.)** The compiled swf file opens. Here is an overview of the user interface elements:


Input text fields. All Single line, with wordWrap=true.

Arrow buttons. All are instances of arrowBtn in the Library.

Blank buttons are here.

Matching Formulas to Data -- Template 1

This applets allows you to enter any number of points and a formula for a function f(x). The formula may contain parameters a, b, c, and d. Use ordinary syntax, for example: a*sin(pi*x*b)-e^x+c*x. You can then easily change the parameters to find a function which best approximates your data.

HOW TO **SYNTAX**

Mouse over HOW TO and SYNTAX for complete instructions.

f(x)= a*cos(b*x-d)+c

GRAPH

a= 7.5 b= -0.5

c= -5.5 d= 0

ymax 10

ymin -14

-10 xmin xmax 10

Least Squares Error = 36.229048015873

POINTS:

(1,0.5)(2,0.3)(3.5,-3)(-1,0.5)(-2,0.3)(-3.5,-3)

Do you prefer to enter your points with:

- round parentheses: (1,2)(-3,1.1)
- brackets: [1,2][-3,1.1]
- no parentheses: 1,2,-3,1.1

RESET

A dynamic text field.

The graphing board created programmatically. An instance of the custom class MatchGraphingBoard.

Ordinary buttons. Also created from scratch.

Radio buttons. These are movie clips created from scratch.

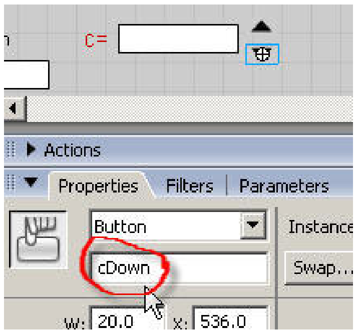
Scrolling buttons created from scratch. Both instances of scrollBtn in the Library.

Picture 2

As we see, the applet contains a number of input text fields and one dynamic text field. The radio clips and scrolling buttons for the points input box were created from scratch. We do not use components in this applet. All up and down arrows residing next to the parameters' input boxes are instances of the same arrowBtn in the Library. We also have two ordinary looking buttons, GRAPH and RESET. They were created from scratch in our article about a function grapher mentioned above. We copied them from a function grapher fla file.

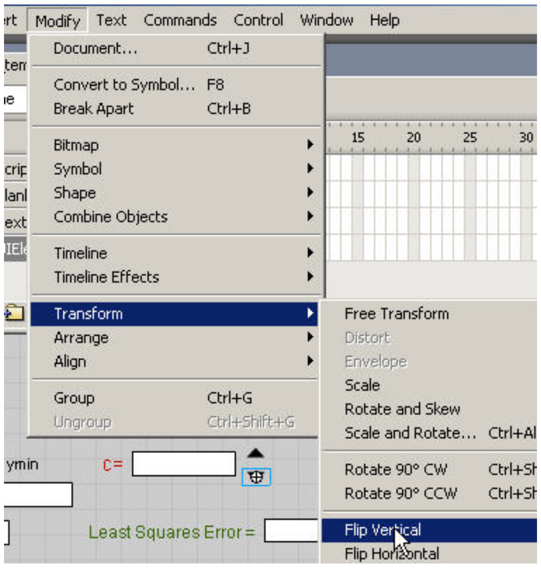
The most prominent element -- the graphing board -- is created programmatically at runtime. It is an instance of the custom class, MatchGraphingBoard, contained in the nested folders edu→edu→uriship→math→match.

Familiarize yourself with the functionality of the applet. Then, close the swf file and go back to the fla file. Before we can discuss the code, please take note of the instance names of all the user interface elements (except for static text boxes). In our ActionScript we will refer to those elements using their instance names. To see each name, select an element and check the Instance Name field in the Properties panel:



Picture 3

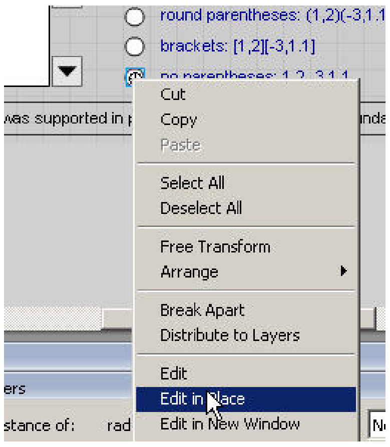
Up and down arrow buttons next to each parameter box are all instances of arrowBtn in the Library. As you see, some of the instances have been flipped upside-down. You can do it using the Transform menu item:



Picture 4

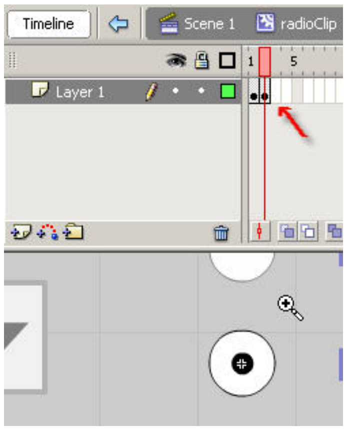
You can apply Transform to an instance of a button or a movie clip without affecting other instances.

To see how the radio clips are constructed, select one of them, right-click on it, and choose Edit in Place from the menu that opens:



Picture 5

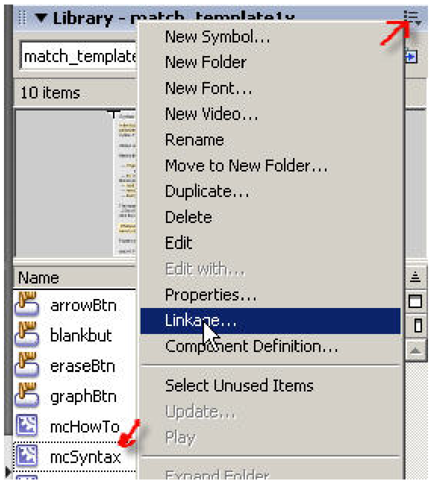
You enter the clip's timeline. As you see, the clip has two frames. Frame 1 contains an empty disk, Frame 2 a circle with a black dot inside:



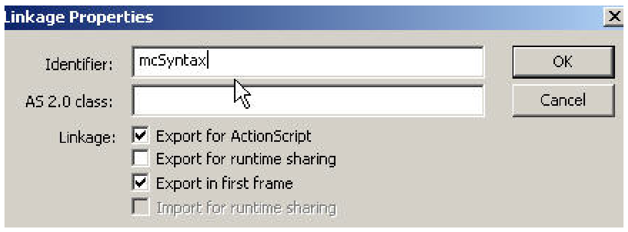
Picture 6

Go To Edit → Edit Document in the uppermost menu. This brings you back to the main timeline.

The last comment we should make before we delve into the code, concerns the two clips, mcSyntax, mcHowTo in the Library and the blank buttons. The clips are present in the Library. They are clearly the clips which appeared and disappeared as we moused over the "How To" and the "Syntax" text in the compiled movie, yet we do not see instances of these clips on the Stage. That is because we will attach them at runtime. For the latter to be possible, we have established the linkage from each of the two clips in the Library to ActionScript. Recall that to do that, you click on the tiny icon at the upper right corner of the Library window, select Linkage from the menu that opens. Then select Export to ActionScript in the dialog box and choose a name for your clip. Only then will you be able to access the clip at runtime:



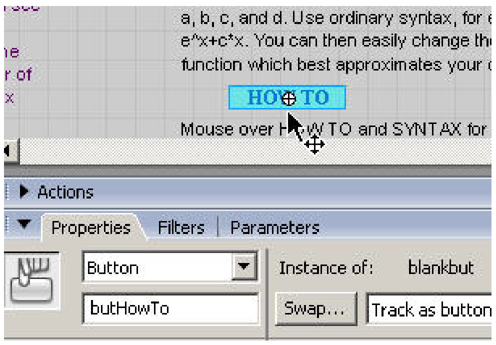
Picture 7



Identifier: mcSyntax
AS 2.0 class:
Linkage: [✓ Export for ActionScript]
Export for runtime sharing
[✓ Export in first frame]
Import for runtime sharing

Picture 8

The clips appearing and disappearing as you mouse over the corresponding text are triggered by the two blank buttons that we have placed over the text:


a, b, c, and d. Use ordinary syntax, for
e^x+c*x. You can then easily change th
function which best approximates your
HOW TO
Mouse over HOW TO and SYNTAX for
Actions
Properties
Filters
Parameters
Swap...
Track as button

Button
butHowTo
Instance of: blankbut

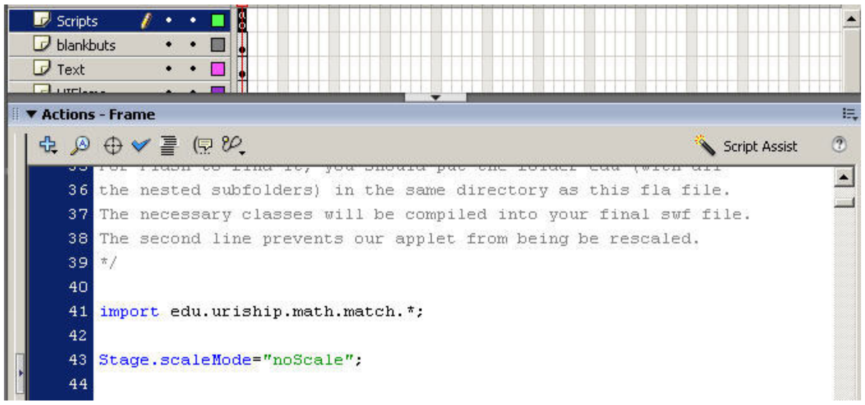
Picture 9

If you want to know how to create a blank button, right-click on the button selected above, and examine its timeline. All frames of the button are empty except for the hit frame. On the other hand, you can always copy and paste the blank button from one file to another, and each instance can be resized without affecting other instances. Thus, one blank button is all you ever need.

We are ready to look at the code.

The Code Behind the Applet

Select the Scripts layer and open the Actions panel. All the code that is needed to make the applet run is there and in the classes contained in the nested folder inside the folder "edu". (The classes are not precompiled. You can open any of the class files in Flash and examine or alter the class code as well.)



```
36 the nested subfolders) in the same directory as this fla file.
37 The necessary classes will be compiled into your final swf file.
38 The second line prevents our applet from being be rescaled.
39 */
40
41 import edu.uriship.math.match.*;
42
43 Stage.scaleMode="noScale";
44
```

Picture 10

Comments are contained between /*..*/ and appear in light gray. As you see, the code is exhaustively commented, almost line-by-line which should make the code very easy to follow. In this guide, we will focus only on those parts that you may want to change in order to customize the applet, and on the parts that are new to our *Flash Tools for Developers* series.

The package imported on line 41 contains MathParser, RangeParser, PointsParser, MatchGraphingBoard, and a few other custom classes needed in the applet. As we mentioned, for Flash to find the classes, you must put the folder edu (with all the nested subfolders) in the same directory as your fla file. The necessary classes will be compiled into your final swf file. The second line prevents our applet from being rescaled which is always a good idea in an applet where so much depends on pixel calculations.

```
import edu.uriship.math.match.*;

Stage.scaleMode="noScale";
```

On line 57, we evoke the constructor of the MatchGraphingBoard class and store the instance created in the variable "board". The constructor creates a square board where all your graphs and points will appear. The parameters passed to the constructor are: x and y coordinates of the upper left corner of the board (relative to the target clip), the size of the board in pixels, the target movie clip in which the board will reside, ("this" below refers to the main movie), and finally the depth in the target clip at which the board will reside. You can change the position and the size of the graphing board by changing the parameters.

```
var board:MatchGraphingBoard=new MatchGraphingBoard(20,20,320,this,1);
```

For example, change "320" to "150" and test the movie. Everything works the same, except that the graphing board is much smaller. Of course, to obtain a pleasing appearance you would want to reposition the range boxes and other elements to fit the new dimensions of the graphing board. The elements created manually you move manually in the fla file. Elements created programmatically (for example the syntax error display box or the coordinates display box) can be manipulated using the corresponding instance methods of MatchGraphingBoard class. It is the instance “board” of the class that controls those boxes. The instance is also responsible for plotting the function f(x) and the points entered by the user, for drawing axes, and for changing the x and y ranges. Below we will see how to use methods of the class to change all the colors and the appearance of the graphing elements controlled by “board”.

As we already mentioned, the movie clip that explains the syntax that MathParser understands, and the movie clip that explains how to enter data points have been created by hand and stored in the library with linkage to ActionScript. We are attaching the clips at runtime and store them in variables mcSyn and mcHow:

```
var mcSyn:MovieClip=this.attachMovie("mcSyntax","syntax",2);

var mcHow:MovieClip=this.attachMovie("mcHowTo","howto",3);
```

A few lines later, we will set their visibility to false. If you want to edit either one of the clips, open the Library panel by going to Window (in the menu at the top) and checking Library. In the Library, select mcSyntax item and drag its image from the Library window to the Stage. Right-click on the clip on the Stage and go to Edit in Place. After you complete your edits, select Edit Document from the Edit menu to go back to the main Stage and delete the clip from the Stage.

The next important class that we will instantiate is the MathParser class. We store the instance in the variable procFun:

```
var procFun:MathParser=new MathParser(["x","a","b","c","d"]);
```

MathParser will be used for compiling and evaluating the user's formula for the function f(x) to be graphed. The constructor takes an array of strings as a parameter. The array contains the names of variables that are allowed and will be recognized by the compiling method of our MathParser, procFun.doCompile(*inputstring*). It is important to remember the array passed to the constructor as the evaluator method, procFun.doEval(...,[..]), will expect an array of values for the variables passed to it in the same order as its second parameter. In this case, values for x, a, b, c, d. For example, procFun.doEval(...,[xmin+i*xstep,nA,nB,nC,nD]). MathParser is case insensitive. The thing to remember is to enter variables as strings, i.e. “x” and not x. In the new improved version of the parser that comes with this article, an output window will alert you if you forget the quotations marks.

We are also creating an instance of the custom class RangeParser which we will use for compiling and evaluating user's input for x and y ranges. The constructor takes no parameters. The only input besides numeric that is allowed in the range boxes are expressions containing the constant pi, e.g. 2*pi or 3*pi/3, etc. The RangeParser knows it.

```
var procRange:RangeParser=new RangeParser();
```

A new class in this article is the class PointsParser that we will use for compiling and evaluating user's data points input. The constructor takes no parameters. The parser has three compiling methods which allow the user to choose a format for entering points: (1,2)(3,4) etc., or [1,2][3,4] etc., or 1,2,3,4. This gives a greater flexibility for the user to copy and paste data points from different sources. For example, data points stored in an Excel sheet can be saved in the comma-delimited format without parentheses.

```
var procPoints:PointsParser=new PointsParser();
```

There are many other global variables in the script. See the comments in the script for explanations. Below we discuss only those variables which are related to the new functionality; that is, to plotting points and animating graphs when parameters change.

The eight Boolean variables:

```
var aUpPressed:Boolean=false;

var aDownPressed:Boolean=false;

var bUpPressed:Boolean=false;
.
.
.
```

and so on, store the information if any of the up and down arrow buttons corresponding to the parameters a, b, c, d is pressed. nA, sA etc. variables store the current numerical and string values of the parameters a, b, c, d. They are all initialized to 1.

```
var sA:String="1";

var nA:Number=Number(sA);

var sB:String="1";

var nB:Number=Number(sB);
.
.
.
```

The variable whichDelim remembers the user's choice of the format in which to enter points. We initialize it to round parentheses.

```
var whichDelim:String="Round";
```

We set the initial settings for the radio clips corresponding to the initial choice of the format for data points to the round parenthesis. As we saw above, the radio clips were created manually. Frame 1 corresponds to radio clip appearing unselected, Frame 2 to selected.

```
mcRadRound.gotoAndStop(2);

mcRadSquare.gotoAndStop(1);

mcRadNone.gotoAndStop(1);
```

When setting properties of input boxes, we should remember to choose Single Line option (either programmatically or in the Properties panel) and set wordWrap to true. Those settings allow multi-line input display while preventing hard line breaks which ruin the input string.

```
FunBox.wordWrap=true;
.
.
.
LSBox.wordWrap=true;
```

The next two comments are self-explanatory. They describe two of the many methods of the MatchGraphingBoard class.

```
/*
board.disableCoordsDisplay();
board.enableCoordsDisplay();
These two methods of the GraphingBoard class enable or disable
the display of x,y coordinates as the user mouses over the board.
Since we want here the "enable" setting which is the default,
we do not need to evoke these methods.
*/

/*
board.disableErrorBox();
board.enableErrorBox();
These two methods of the GraphingBoard class enable or disable
the display of an error box in which errors in syntax can be
displayed for the user. The "enable" setting is the default,
and we want the error box to be enabled so
we do not need to evoke these methods. When enabled, board has
a property board.ErrorBox which is a dynamic text field. We will
send there all error messages.
*/
```

The next series of instance methods of MatchGraphingBoard (used with our instance “board”) set the color of the border and the background for the graphing board, the formatting and the position of the syntax error and the coordinate boxes, and the color for the axes. In this applet, we use default colors, so we evoke the methods for illustration purposes only. Note that all position coordinates are in pixels and are relative to “board”, and all colors are passed in their hex form, e.g. 0xFFFFFF. You can change the color codes to customize the appearance of the grapher. For example, if you change the lines:

```
board.changeBorderColor(0x000000);

board.changeBackColor(0xFFFFFF);
```

to:

```
board.changeBorderColor(0xFFFFFF);

board.changeBackColor(0x000000);
```

Your board will be black with white border. Similarly, you can change sizes, positions, and colors for the error and coordinates boxes. Their positions are relative to the instance “board” but they do not have to be physically located within the graphing board.

```
/*
We are setting the format for board.ErrorBox. The parameters determine:
background color, border color, text color, and text size. Again,
we could skip the next line as we are choosing the default values.
*/

board.setErrorBoxFormat(0xFFFFFF,0xFFFFFF,0x000000,12);

/*
We are setting the size and position for board.ErrorBox.
The parameters determine: width, height, and the position of the upper
left corner (relative to board), all in pixels.
*/

board.setErrorBoxSizeAndPos(270,150,20,20);

/*
Since coordinate display is enabled, in the next two lines
we are setting the format, size and position of the coordinate display box.
The parameters in order of appearance give: background color,
border color, text color, text size, width, height, and the
xy coordinates relative to board.
*/

board.setCoordsBoxFormat(0xFFFFFF,0xFFFFFF,0x000000,12);

board.setCoordsBoxSizeAndPos(60,40,20,270);

/*
If we wanted the user to be able to draw on the graphing board with the
mouse, we would call the method
board.enableUserDraw(0x006600,1);
The method takes as parameters the color of the user's sketch and the
thickness of the line. We choose not to enable the user to draw on the board.
*/

/*
Setting color for axes. Since the background of our board is white,
we are setting axes color to black. Since black is the default,
we could skip the next line.
*/

board.setAxesColor(0x000000);
```

The next three methods allow you to choose the size, the color, and the style in which user-defined data points will be displayed. The available styles are Outline, Filled, and Cross. Change the settings below to see what they look like.

```
board.setPointSize(2);

board.setPointColor(0x0000FF);

board.setPointStyle("Outline");
```

The most important function in the script is the makeGraphs function. The function is evoked when the user clicks GRAPH button or if the user presses one of the parameters' up or down arrows. The function makeGraphs parses the input for f(x) and for the x, y ranges, calls errors, sends them to the error box, and produces the graph of f(x) by calling the appropriate board methods. The function also parses the user's input for the points and plots the points. The code within the function is carefully commented and easy to follow. Below, we only highlight the portions of the code within the function makeGraphs which are related to using MathParser, PointsParser, and drawing methods of MatchGraphingBoard class. We skip many local variables and secondary steps related to range parsing, checking entries in the range and parameters' boxes, and other minor tasks.

```
function makeGraphs():Void {
     .
     .
     .

   /*
   The string variables to store the user's formula for the function
   to be graphed and coordinates of the data points entered:
   */

    var sFunction:String="";

    var sPoints:String="";


      /*
      The next variable will store functional values (before
      converting to pixels) of points on the graph of f(x) obtained
      after parsing and evaluation. Those points will be joined by lineal
      elements to produce the graph of f(x).
      */

      var fArray:Array=[];


      /*
      MathParser.doCompile method returns a datatype CompiledObject
      (defined by one of the classes in the package). Any instance
       of CompiledObject has three propetries: CompiledObject.PolishArray
       which represents a mathematical formula in a form
       suitable for evaluation, CompiledObject.errorMes which contains
       a string with an error message should a mistake in syntax be found,
       and, finally, CompiledObject.errorStatus which is 1 if an error
       is detected and 0 otherwise. Below we create three variables to store
       the results of compiling the user's formulas for the function
       f(x).
      */

      var compObj:CompiledObject;

      /*
      PointsParser compiling methods return a datatype PointsObject
      (defined by one of the classes in the package). Any instance
       of PointsObject has three propetries: PointsObject.PointsArray
       which represents the coordinates of the points to be plotted,
       PointsObject.errorMes which contains a string with an error message
       should a mistake in syntax be found, and, finally,
       PointsObject.errorStatus which is 1 if an error
       is detected and 0 otherwise. Below we create a variable to store
       the results of compiling the point data entered by the user.
      */

      var pointsObj:PointsObject;
      .
      .
```

```
.

/*
We clear the graphing board.
*/

board.eraseGraphs();

board.clearPoints();
.
.
.
/*
  Telling board what x and y ranges are after parsing
   the user's entries in the x and y range boxes.
 */

board.setVarsRanges(xmin,xmax,ymin,ymax);

// Drawing axes.

board.drawAxes();

.
.
.

// We compile the formula which is present in the f(x) input box.

if(sFunction.length>0){

/*
Evoking our MathParser doCompile method
to compile the f(x) formula entered by the user.
Recall that the instance of MathParser created above is called
procFun. If an error is found during compiling,
a message is sent to board.ErrorBox and the function quits.
*/

compObj=procFun.doCompile(sFunction);

if(compObj.errorStatus==1){

      board.ErrorBox._visible=true;

      board.ErrorBox.text="Error in f(x). "+compObj.errorMes;

      return;

}

/*
If no error is found we create an array of points, fArray, (a local
variable), fArray is used to create the graph of f(x). Each entry in
the array consists of a pair of [x,y] values. x values are determined
by starting from xmin and adding step-by-step the value which brings us
to the next point on the x axis. This value is stored in a local
variable xstep and depends on the number of points chosen at the
beginning of the script to draw each graph and the x range.

To obtain the corresponding values of y, we evoke the procFun.doEval
method. Observe that the method requires two parameters. The first
parameter has to be a PolishArray of a compiled expression.
In the case below, it is comObj.PolishArray which gives a compiled
```

```
       form of the formula for f(x). The second parameter
       of procFun.doEval method is an array of values for variables recognized
       by our instance of MathParser. In our case, it is an array containing
       the current values for x, a, b, c, d:

        procFun.doEval(compObj.PolishArray,[xmin+xstep*i,nA,nB,nC,nD]).

       The method gives us the y values corresponding to consecutive x values.
       */

       for(i=0;i<=xpoints;i++){

fArray[i]=[xmin+i*xstep,
              procFun.doEval(compObj.PolishArray,[xmin+xstep*i,nA,nB,nC,nD])];
       }

             /*
             Observe, that the array of points created above contains
             Functional values. Those values will be converted to pixel values
             by board. Namely, by board.drawGraph method evoked below. The
             method takes the following parameters: a positive integer which
             will be remembered as the number of the graph. This integer will
             determine the depth of the graph internally within board. Hence,
             no two different graphs should have the same depth. The second
             parameter contains the array of points on the graph of f(x),
             fArray. The pixel equivalents of the points
             will be joined by lineal elements to create the graph of f(x).
             The last parameter determines the color of the graph. You may
             want to change the last parameter to customize the appearance of
             your grapher.

           */

           board.drawGraph(1,fArray,0xFF0000);

       }

       /*
       The string holding data points is being retrieved, and, if present,
       compiled. We choose the compiling method of PointsParser that
       corresponds to the delimiter chosen by the user.
       */

       sPoints=PointsBox.text;

       if(sPoints.length>0){

             if(whichDelim=="Round"){

       pointsObj=procPoints.parsePointsRound(sPoints);

             }

             else if(whichDelim=="Square"){

       pointsObj=procPoints.parsePointsSquare(sPoints);

             } else {

                   pointsObj=procPoints.parsePointsNone(sPoints);

             }

       if(pointsObj.errorStatus==1){
             board.ErrorBox._visible=true;

             board.ErrorBox.text=pointsObj.errorMes;

             return;

       }

       /*
       If no error is found during compilation, the points are plotted and the
       least square error calculated. We use the method drawData. ypoint
       and yfun are local variables introduced for convenience.
       */

       board.drawData(pointsObj.PointsArray);

       for(i=0;i<pointsObj.numPoints;i++){

             ypoint=pointsObj.PointsArray[i][1];

             yfun=procFun.doEval(compObj.PolishArray,
                           [pointsObj.PointsArray[i][0],nA,nB,nC,nD]);

             LSError+=Math.pow(ypoint-yfun,2);

       }

           LSBox.text=String(LSError);

       }

 }
```

The last portion of the script that we want to look at is related to animating the graph of f(x) when the user presses up and down arrows next to parameters' boxes. First we define twelve callback functions corresponding to the event handlers "onPress", "onRelease", and onReleaseOutside". The callback functions are setting the Boolean variables aUpPressed, aDownPressed etc. to true or false depending if the user presses or releases the corresponding arrows.

```
aUp.onPress=function():Void {

      aUpPressed=true;


};

aUp.onRelease=function():Void {

      aUpPressed=false;
};

aUp.onReleaseOutside=function():Void {

      aUpPressed=false;
};
.
.
.
```

The mechanism which creates the animation effect relies on the event handler "onEnterFrame". In our case, we use "onEnterFrame" for the main movie referred to as 'this'. Instead, we could use "*MovieClip*.onEnterFrame" for any movie clip present on the Stage.

The event handler's 'onEnterFrame' callback function is executed at the frame rate of our movie which is set to 12 fps. That is what gives us the animation effect when the user presses the parameters' arrow buttons. Alternatively, you could use ActionScripts's global function:

setInterval(*name of a function*, *time in milliseconds*)

to have a block of code executed repeatedly at set time intervals. The advantage of the latter approach is that setInterval works with the global function updateAfterEvent() which forces a screen refresh between frames. In our applet, we choose this.onEnterFrame tool.

```
this.onEnterFrame=function():Void {

      if(aUpPressed){

      nA+=0.5;
      sA=String(nA);
      aInput.text=sA;
      makeGraphs();


      }

//We repeat similar blocks of code for other up and down arrows.

      .
      .
      .

};
```

The Description of Custom Classes

The package edu.uriship.math.match contains seven custom classes: CompiledObject, MathParser, MatchGraphingBoard, PointsParser, PointsObject, RangeParser, RangeObject. Here is the description of each class.

- **CompiledObject**

This simple class defines a datatype that is returned by MathParser. The constructor takes no parameters.

```
      var compObj:CompiledObject = new CompiledObject();
```

Every instance of CompiledObject has three properties:

`compObj.PolishArray` -- an array. When compObj is returned by MathParser's doCompile method, the property represents a parsed mathematical expression in the Polish notation. Default value [].

`compObj.errorMes` -- a string. When compObj is returned by MathParser's doCompile method, the property represents a specific syntax error message. Default value "".

`compObj.errorStatus` -- a number 0 or 1. When compObj is returned by doCompile, 0 corresponds to no error found, 1 to error found. Default value 0.

Within the two templates, we only use the instances of the class which are returned by MathParser's doCompile method.

- **MathParser**

This class is the engine behind parsing the user's input. The constructor takes an array of strings as a parameter. For example:

```
var procFun:MathParser = new MathParser(["x","a","b","c","d"]);
```

The array of strings represents names of variables that will be recognized by the instance of MathParser. In the example above as well as in our templates, we use five variables "x", "a", "b", "c", and "d". There can be any number of variables, e.g.: new MathParser(["x","y","z"]), and they can have names longer than one letter. If you do not want your instance of the parser to allow variables, enter the empty array into the constructor: new MathParser([]). Constants "pi" and "e" are automatically recognized and evaluated by the parser; do not enter them into the constructor.

Every instance of MathParser, in our script we call the instance we create procFun, has two methods:

`procFun.doCompile(`*`string`*`)` -- this method takes a string (typically a string entered by the user) and returns an instance of CompiledObject. Earlier we saw, the method used within makeGraphs function. The CompiledObject returned was stored in a local variable compObj. If there were no errors found during compilation

```
compObj.PolishArray
```

was sent to the evaluator method, doEval, of MathParser. The method is discussed next.

`procFun.doEval(`*`array, array`*`)` -- this method takes two arrays as parameters. For the method to do what you want it to do, the first array has to represent a mathematical expression in the Polish notation returned by the doCompile method, the second array provides values of the variables recognized by the parser listed in the same order as the order in which the variables were passed to the MathParser constructor. In our templates, there are five variables, "x", "a", "b", "c", "d", so the second array has five entries. In our script, it looks as follows:

```
     procFun.doEval(compObj.PolishArray,[xmin+xstep*i,nA,nB,nC,nD]);
```

(xmin+xstep*i, nA, nB, nC, and nD are numerical variables which were defined earlier in the script. They represent current values for x, a, b, c, and d.)

The complete list of functions that MathParser recognizes as well as all the syntax rules are described in a movie clip that appears in each of the templates when the user mouses over the SYNTAX button.

It should be noted that MathParser in this package differs slightly from the MathParser class in the packages which came with the authors' previous articles. Any of the earlier versions of parser would do the job just fine. In this version, we made the parser case insensitive as far as variables go. Similarly as in our *Flash Tools for Developers: 3D Function Grapher* article, the evaluator method is slightly improved for speed. Also, we added a message to the programmer which will appear in the output window should a name of a variable be entered into the constructor without quotation marks. This seems to be a common mistake when using the class.

- **PointsObject**

This simple class defines a datatype that is returned by all three compiling methods of PointsParser. It is very similar to CompiledObject. The constructor takes no parameters.

```
      var instanceName:PointsObject = new PointsObject();
```

Every instance of PointsObject has four instance properties:

`instanceName.PointsArray` -- an array. When the instance is returned by a PointsParser's compiling method, the property represents the array of two-element arrays: [[x1,y1],[x2,y2]….] which give x and y coordinates of data points. Default value [].

`instanceName.errorMes` -- a string. When the instance is returned by a PointsParser's compiling method, the property represents a specific syntax error message. Default value "".

`instanceName.errorStatus` -- a number 0 or 1. When `instanceName` is returned by any of the three PointsParser's compiling methods, 0 corresponds to no error found, 1 to error found. Default value 0.

`instanceName.numPoints` -- a number of points, that is, the length of the PointsArray. Default value 0.

Within the two templates, we only use the instances of the class that are returned by PointsParser's compiling methods.

- **PointsParser**

The PointsParser is a simple utility for parsing the user's data points input. The constructor takes no parameters. For example:

```
var procPoints:PointsParser=new PointsParser();
```

Each instance has three compiling methods. Each of the three methods takes a string as a parameter and returns an instance of PointsObject:

`instanceName``.parsePointsRound(`*`string`*`)`

`instanceName``.parsePointsSquare(`*`string`*`)`

`instanceName``.parsePointsNone(`*`string`*`)`

The first method parses the user's data points entered with round parentheses as a delimiter, e.g:

(1,2)(3,-4)(3.5,5) or   (1,2),(3,-4),(3.5,5)

(Commas between points are optional.)

The second method parses the user's data points entered with brackets as a delimiter, e.g:

[1,2][3,-4][3.5,5] or   [1,2],[3,-4],[3.5,5]

(Again, commas between points are optional.) The last method parses points entered as a sequence of coordinates without a delimiter between points:

1,2,3,-4,3.5,5

The instance of PointsObject returned, say pointsObj, reflects if the error during compilation was detected. If yes, pointsObj.errorStatus=1, pointsObj.errorMes is a string alerting the user as to where the error is, pointsObj.PointsArray=[]. If no error is detected, pointsObj.errorStatus=0, pointsObj.errorMes="", pointsObj.PointsArray=[[x1,y1],[x2,y2],….] gives the coordinates of the data points.

- **MatchGraphingBoard**

This class is responsible for creating all visual elements within the graphing board, including the syntax error box and coordinates display box. The constructor takes five parameters: the positon in pixels of the upper left corner relative to the target movie clip, the size of the square graphing board that will be drawn, the target movie clip ("this" refers to the main movie), and the depth within the target clip. For example:

```
var board:MatchGraphingBoard=new MatchGraphingBoard(20,20,320,this,1);
```

The methods of the class are used and commented exhaustively in the templates' scripts as well as in the sections above. We list them below for the sake of completeness.

- Instance Properties

  `instanceName.ErrorBox` -- a dynamic text field. By default located within the graphing board in the upper half of it. By default the background and border colors of the box are both white, text color black, text size 12 points. The script in each of the two templates displays error messages to the user in this field. For example:

  ```
  board.ErrorBox.text="Error in f(x). "+compObj.errorMes;
  ```

- Instance Methods

For simplicity, we list the methods below with sample values for parameters. Unless said otherwise, each method returns nothing. In each template's script, the instance of the MatchGraphingBoard is called "board".

`instanceName.setBackColor(0xFFFFFF)` -- determines the color of the graphing board background. The color is passed as a parameter in hex. If the method is not called, the color of the background will be white by default.

`instanceName.setBorderColor(0x000000)` -- determines the color of the border of the graphing board. The color is passed as a parameter in hex. If the method is not called, the color of the border will be black by default.

`instanceName.enableErrorBox()` -- enables syntax error box display (default).

`instanceName.disableErrorBox()`-- disables syntax error box display.

`instanceName.setErrorBoxFormat(0xFFFFFF,0xFFFFFF,0x000000,12)` -- determines the colors of the error box's background, border, the color of the font, and the size of the font. If the method is not called, the default values are: white, white, black, 12.

`instanceName.setErrorBoxSizeAndPos(270,150,20,20)` -- determines the width, the height, and the x and y coordinates (all in pixels) of the error box. The position is with respect to the upper left corner of the graphing board. If the method is not called, the default values will position the Error box over the top half of the graphing board.

`instanceName.enableCoordsDisplay()` -- enables display of x and y coordinates as the user mouses over the graphing board (enabled by default).

`instanceName.disableCoordsDisplay()`-- disables coordinates display.

`instanceName.setCoordsBoxFormat(0xFFFFFF,0xFFFFFF,0x000000,12)` -- determines the colors of the coordinates box's background, border, the color of the font, and the size of the font. If the method is not called, the default values are: white, white, black, 12.

`instanceName.setCoordsBoxSizeAndPos(60,40,20,270)` -- determines the width, the height, and the x and y coordinates (all in pixels) of the coordinates display box. The position is with respect to the upper left corner of the graphing board. If the method is not called, the default values will position the coordinates box within the graphing board, near its lower left corner.

`instanceName.setAxesColor(0xCCCCCC)` -- determines the color in which the x and the y axes will be drawn. Default: black.

`instanceName.setPointColor(0x0000FF)` -- determines the color in which the data points will be drawn. Default: blue.

`instanceName.setPointSize(5)` -- determines the size of data points. The size has a slightly different meaning for each of the three styles of data points. Default: 2.

`instanceName.setPointStyle("Cross")` -- determines the style of data points. There are three styles available: "Outline", "Filled", and "Cross". Default: "Outline".

`instanceName.setVarsRanges(-10,10,-10,10)` -- sets the x and y ranges for the graphing board. The method should be called before any of the three next methods.

`instanceName.drawAxes()` -- draws the x and the y axes.

`instanceName.drawGraph(1,fArray,0xFF0000)` -- draws the graph of a function based on an array, fArray, of [x,y] points (in functional terms) along the graph of the function to be graphed. The first parameter is an integer denoting the number of the graph. No two graphs should have the same number as the number determines the depth of the graph within the instance of the MatchGraphingBoard. The last parameter is the color in which the graph will be drawn. The method returns an array of two-element arrays which represent pixel equivalents of the [x,y] points in fArray. In this applet, we do not use the return value of the method.

`instanceName.drawData(Array)` -- given an array of two-elements arrays (in our applet it is pointsObj.PointsArray) the method plots data points. (The coordinates of data points are passed in terms of their functional values. The method translates them to their pixel equivalents relative to the graphing board.)

`instanceName.eraseGraphs()` -- the method erases graphs of functions present on the graphing board.

`instanceName.clearPoints()` -- the method erases drawings of data points present on the graphing board.

`instanceName.isNotLegal(a:Object)` -- the method determines if "a" is a finite number. Returns "true" if "a" isn't a legal number, "false" otherwise.

Instance methods not used in our templates

`instanceName.enableUserDraw(0xFFFF00,1)` -- enables the user to draw on the board using mouse. The parameters determine the color and the thickness of the line.

`instanceName.eraseUserDraw()` -- erases the user's drawings.

`instanceName.disableUserDraw()` -- prevents the user from drawing on the board.

`instanceName.getBoardSize()` -- returns the size of the board.

`instanceName.getPointSize()` -- returns the size of data points.

`instanceName.getVarsRanges()` -- returns the current x and y ranges.

`instanceName.isDrawable(a:Number)` -- determines if a pixel value, "a", for the x or y coordinate is too large to be drawn without causing the "wrapping at infinity" effect. Returns true or false.

`instanceName.isLegal(a:Object)` -- the method determines if "a" is a finite number. Returns "false" if "a" isn't a legal number, "true" otherwise.

`instanceName.destroy()` -- the method removes listeners, deletes movie clips created by *instanceName*, etc. You should evoke the method before deleting the variable storing the instance (in our templates it is the variable "board") should you need to delete it.

- **RangeObject**

This simple class defines a datatype that is returned by the parseRange method of RangeParser. It is similar to CompiledObject. The constructor takes no parameters.

```
var rangeObj:RangeObject = new RangeObject();
```

Every instance of RangeObject has three public properties:

`rangeObj.Values` -- an array. When rangeObj is returned by RangeParser's parseRange method, the property represents the four numerical values for xmin, xmax, ymin, ymax. Default value [].

`rangeObj.errorMes` -- a string. When rangeObj is returned by parseRange method, the property represents a specific syntax error message. Default value "".

`rangeObj.errorStatus` -- a number 0 or 1. When rangeObj is returned by RangeParser, 0 corresponds to no error found, 1 to error found. Default value 0.

Within the two templates, we only use the instances of the class that are returned by parseRange method of RangeParser.

- **RangeParser**

The RangerParser is a simple utility for parsing the user's input in the range boxes. We need RangerParser to allow inputs containing "pi" like pi/2, 3*pi/2 2*pi etc., in addition to numerical inputs. The constructor does not take any parameters:

```
var procRange:RangeParser = new RangeParser();
```

Each instance has only one method, parseRange:

```
procRange.parseRange(string,string,string,string).
```

The method takes four strings as parameters and returns an instance of RangeObject.

Suggestions for Improvements

An attractive direction for expanding the capabilities of the Matching Formulas to Data applets presented in this article is adding buttons that plot regression curves for a given collection of data points. For example, adding a button that plots the regression line and displays its equation. Formulas for the linear regression are not complicated, so it wouldn't be hard to add such a capability. However, calculating other types of regression curves is a bit more involved. We hope that our readers will take over this task.

Recommended Reading

The complete Flash 8 documentation, including Getting Started with Flash, Using Flash, Learning ActionScript 2.0 in Flash, ActionScript 2.0 Language Reference, Using Components in Flash, and more can be downloaded for free from the Adobe/Macromedia site:

http://www.adobe.com/support/documentation/en/flash/

(After you open the page, click on the link Download Complete Flash 8 Documentation.)

A few books that we have found immensely helpful are listed below.

To become familiar with the Flash's authoring environment:

1. Katherine Ulrich, Macromedia Flash 8 for Windows and Macintosh: Visual QuickStart Guide, Pearson Education, 2006.

2. Robert Reinhardt and Snow Dowd, Macromedia Flash 8 Bible, Wiley Publishing, 2006.

To learn ActionScript 2.0 (which hasn't changed between Flash MX 2004 and Flash 8 except for some classes and functions added):

3. Colin Moock, Essential ActionScript 2.0, O'Reilly, 2004.

4. Macromedia Flash MX 2004 ActionScript 2.0 Dictionary, Macromedia Press, 2003.

5. Robert Reinhardt and Joey Lott, Flash MX 2004 ActionScript Bible, Wiley Publishing 2004.

6. Colin Moock, ActionScript for Flash MX: The Definitive Guide, Second Edition, O'Reilly, 2003.

The latter book was written for an earlier version, Flash MX, but we still find it to be a great reference.