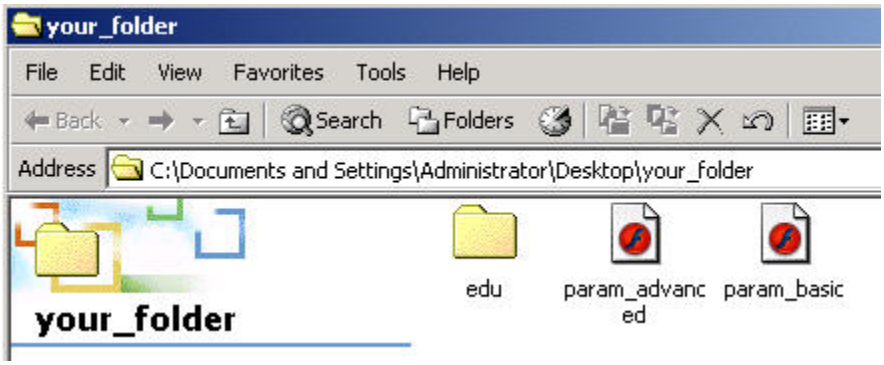
Flash Tools for Developers: Parametric Curves
A Guide

This paper is a companion to the online article at the MathDL Digital Classroom Resources "Flash Tools for Developers: Parametric Curves" by Douglas E. Ensley and Barbara Kaskosz. This paper provides instructions on how to use the two parametric curves templates and the ActionScript classes presented in the article. The templates and the classes discussed below can be downloaded from the article through the link param_graph.zip.

Getting Started

Download param_graph.zip file and unzip it in a folder on your computer. You will see a param_graph folder which contains all the files related to the article. These include the folder "edu" which contains all the necessary ActionScript 2.0 classes. The classes reside in a nested sequence of folders: edu→uriship→math→parametric2d. Besides the folder with the classes, you will see the fla files for the two parametric curves templates: param_basic.fla and param_advanced.fla.

You can start working right from the folder param_graph. Simply open one of the two template fla files in Flash MX 2004 Professional or Flash 8 Professional and begin customizing the template. Or, if you prefer a clean working environment you can copy the "edu" folder and one or both 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:

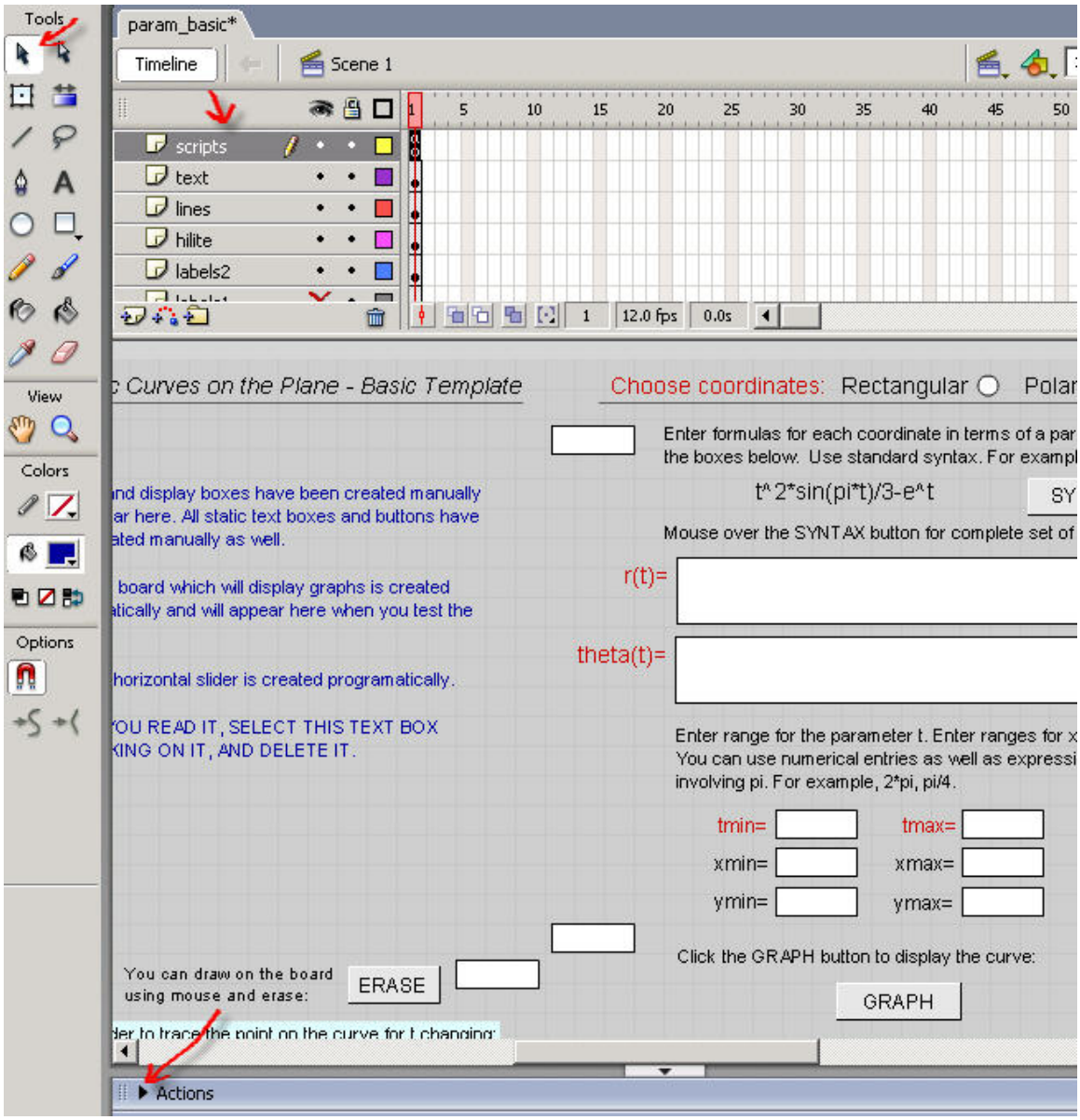


Picture 1

Open your version of Flash, navigate to, and open one of the templates. In this guide, we will work mostly with param_basic template. After you open the template, you may want to save it under a new name as your working file, for example, your_grapher.fla. To do that, go to File→Save As, click on Save As, type the new name in the dialog box, and click save. If you are in Flash 8, a dialog box will appear asking you if you want to convert the file (which was originally created in Flash MX 2004) to Flash 8 format. If you plan to continue working in Flash 8, click "Save". If you are in Flash 2004, no dialog box will appear.

The Two Templates

The template param_basic.fla provides basic functionality necessary for graphing and tracing parametric curves on the plane in rectangular and polar coordinates. The template param_advanced.fla provides additional functionality: setting x and y ranges automatically, choosing between scaling constrained and scaling not constrained options. If you plan on doing much customization or you want to familiarize yourself with the most essential elements of the code first, you should begin with the param_basic.fla template. Here is a screen shot of the open param_basic.fla file:



Picture 2

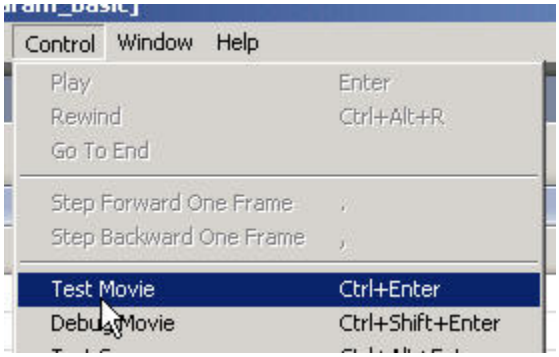
(Most of the screen shots in this article show Flash MX 2004 but everything looks very similar in Flash 8.)

In this article, we assume you are familiar with the basics of Flash authoring environment: the Timeline, the Stage, the Properties and the Actions panels, the Library. We also assume you are familiar with the concepts of text boxes, buttons, movie clips and other elements on the Stage as well as with the idea of their instance names. If you are not familiar with those concepts, you may want to look at the companion guide, fun_grapher.pdf, for our previous article, "Flash Tools for Developers: Function Grapher". It explains all the basics.

Unless you are performing specialized tasks, you should have the Selection Tool selected in the Tools panel marked by the red arrow on Picture 2.

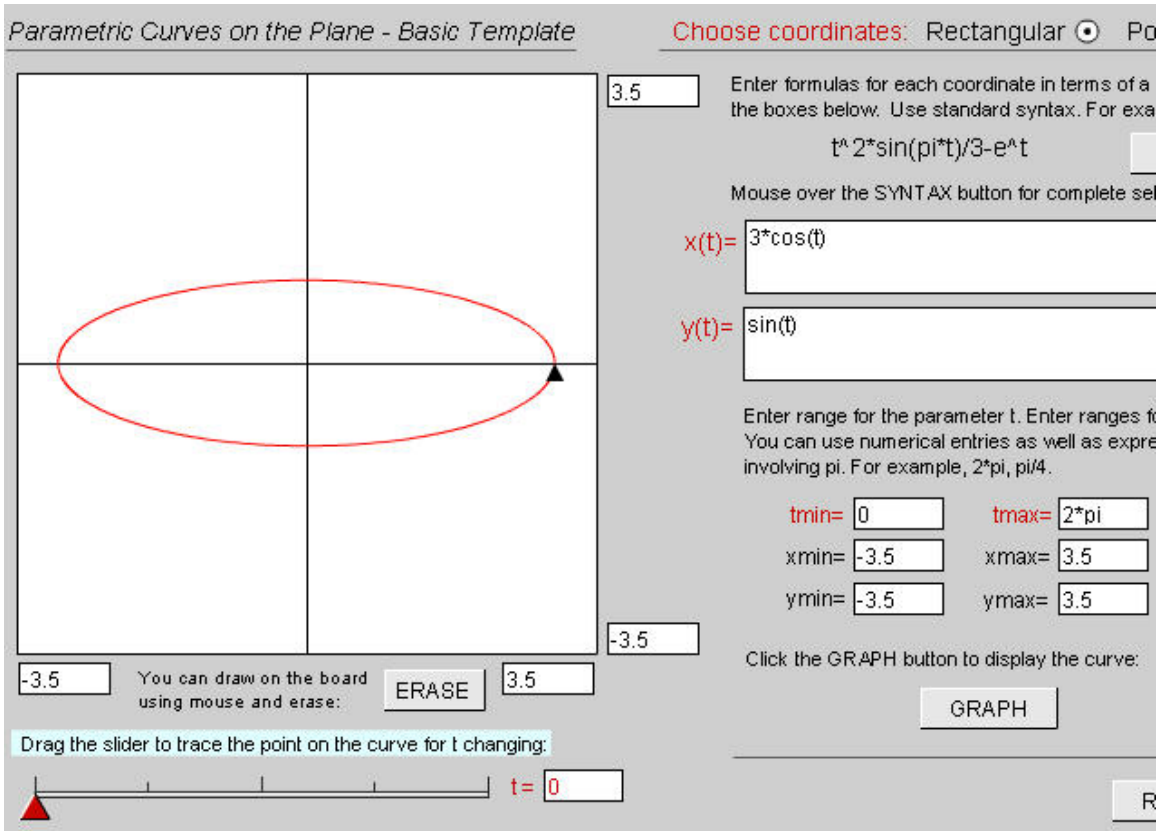
Elements of the param_basic fla File

As you see on the screen shot above, the param_basic.fla contains many elements that were created manually and they are visible as you open the file: input, dynamic, and static text boxes, buttons, and radio buttons (which are actually movie clips used as buttons). The most important elements of the applet, the graphing board as well as a horizontal slider, are not visible as they are created programmatically. To see them and to see how the applet works, go to Control menu item at the top of the page and choose Test Movie.



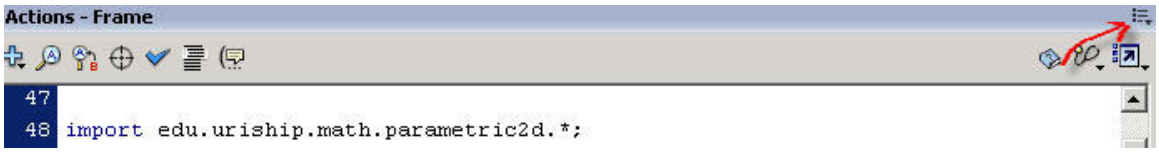
Picture 3

A compiled, swf, file playing in the Flash player appears:



Picture 4

Familiarize yourself with the grapher's functionality, then close the swf file by clicking on the "X" in its upper right corner. You are back in the fla file. All the code that makes the applet run is located on Frame 1 of the "scripts" layer. The layer is marked with a red arrow on Picture 2. Select the layer by clicking on it and then click on the Actions panel name to open it. (The panel is marked by a red arrow on Picture 2 as well.) The Action panel opens revealing all the code:

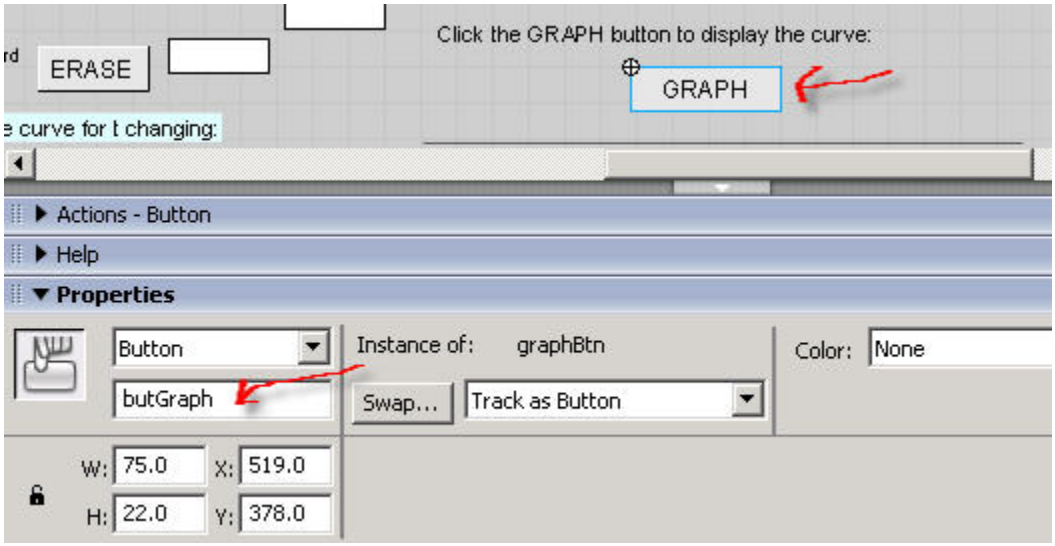


Picture 5

The code you see there, technically attached to Frame 1 of "scripts" layer, contains all the code needed by the applet except for the code contained in the classes in edu→uriship→math→parametric2d. Those classes are imported on line 48. (If line numbers do not appear when you open the Actions panel click on the tiny icon at the upper right corner of the Actions panel, marked with a red arrow on Picture 5, and check View Line Numbers in the menu that drops down.)

In the sections that follow, we will be modifying applet's properties by changing the code. All elements on the Stage are accessed by ActionScript through their instance

names. Let's make sure we know how to see or assign instance names. Close for a moment the Actions panel by clicking on its name and open the Properties panel. Select an element on the Stage by clicking on it. For example, select the GRAPH button. The blue outline around the button appears and the Properties panel displays properties of the button: its size, position, and, most importantly, its instance name butGraph marked by the red arrow below.



Picture 6

In ActionScript 1, it was common practice to attach code directly to objects, like buttons, movie clips, etc. In that case you often didn't need instance names. In ActionScript 2.0, it is more common to keep your code in one place. In this case, you can't access objects if they don't have instance names.

Modifying the Script: Changing Appearance

In this section, we will show how to customize the appearance and the properties of the grapher through simple modifications of the script. Select the "scripts" layer and open the Actions panel. Most of the text you see are comments. Comments are enclosed between /*…*/ or following //, and they appear in faded grey. The comments are designed to walk you step-by-step through the script. For the purpose of our little exercise, scroll to line 62:

```
var board:ParamGraphingBoard=new ParamGraphingBoard(20,39,320,this,1);
```

On this line we are instantiating the custom class, ParamGraphingBoard, contained in the package edu.urishiip.math.parametric2d (the package was imported on line 48) and storing the instance in the variable "board". The instance "board" will create a square graphing board where all your graphs will appear. "board" will also be responsible for creating graphs, for formatting and displaying the error box, the coordinates display box, and for performing other tasks. The constructor of the class, evoked by the word "new", takes several parameters. The first three are responsible for the location and the size of the board within the target movie clip. The target movie clip above is the main movie,

referred to as "this", and "board" is placed on the depth 1 which is specified by the last parameter. "board" is 320 by 320 pixels and its upper left corner is located at the position x=20, y=39. (Recall that coordinates in a fla file are measured in pixels from the upper left corner of the movie, (0,0). "x" increases to the right, "y" increases as you go down.)

For the purposes of this demonstration, we begin our customization by changing colors and properties of some of the elements controlled by "board". Suppose we want a black graphing board with a white border. It takes only a few simple changes to the code.

Scroll to line 188. You see there:

```
board.enableTrace(true);
```

(The method enables tracing functionality. We will talk about it later.) Just below that line (or anywhere else in the script for that matter) place the line:

```
board.changeBorderColor(0xFFFFFF);
```

(As you see, the method takes as a parameter a hex number of a color.) We didn't evoke this method in the original script as the default border color is black and that was what we wanted. Similarly, we didn't evoke the method that causes the background color to change as the color is white by default. To change the background color, below the line that you just typed, add the code:

```
board.changeBackColor(0x000000);
```

Go to Control →Test Movie. As you see, the board is black, with white border. The axes are not visible and neither is the tracing arrow as, by default, they are set to black. We need to change that. Below what you just typed, add lines:

```
board.setAxesColor(0xCCCCCC);
board.changeArrowColor(0xFFFF00);
```

Test the movie now. The axes are visible in light grey and the tracing arrow is yellow.

We are not all set, though. If you mouse over the board, the coordinate display box still has a white background. If you enter a formula for one of the functions with a syntax error, i.e., 3*cos(t)- , and click GRAPH, the error box that appears has a white background. We need to change all that. Close the compiled movie and go back to the fla file.

Scroll to line 340. (From now on, line numbers are approximate as we added some code. You see there:

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

The command sets the background, the border, and the text color and size of the error box. Change it to:

```
board.setErrorBoxFormat (0x000000,0x000000,0xCCCCCC,12);
```

Scroll a few lines down to 358. You see there:

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

Change it to:

```
board.setCoordsBoxFormat (0x000000,0x000000,0xCCCCCC,12);
```

The method takes parameters that control your coordinate display box's background color, border color, text color, and text size.

Test the movie now. Mouse over the board and try to graph "3*cos(t)-". The error box and coordinate display box appear in proper colors. Try to draw on the board with the mouse. Your drawing is in blue which appears too dark. To change that, go back to the fla file and scroll to line approximately 369:

```
board.enableUserDraw(0x0000CC,1);
```

The line instructs "board" to enable the user to draw on the board in blue, with the line thickness 1. Change it to whatever color and thickness you want, for example, light blue:

```
board.enableUserDraw(0xCCFFFF,1);
```

Test the movie now. A user's drawing appears fine. The feature of drawing on the board with the mouse can be shut down by simply not evoking the method above. (The default setting is user drawing disabled). We find the drawing capability useful, especially in classroom demonstrations, when you want to point to some aspect of the graph of a curve.

"board", as an instance of the class ParamGraphingBoard, controls the color of graphs it produces. To change them go back to the fla file and scroll to line approximately 676:

```
locArrowPos=board.drawGraph(dep,fArray,0xFF0000);
```

The method is evoked within a function drawCurve(dep) which has already parsed the user's input for a curve and produced an array of points fArray to be graphed. The first parameter of the method is the number of the graph (it also is the depth of the graph within a movie clip internal to "board"). The last parameter determines the color of the graph, in this case, red. Let's change it to light blue :

```
locArrowPos=board.drawGraph(dep,fArray,0xCCFFFF);
```

or any other color you wish. (The meaning of the left-hand side of the line is explained extensively in the comments within the fla file and in one of the sections below.)

Test the movie, and see if you like your color scheme. Assume yes, and let's move on to customizing the appearance of the horizontal slider.

The horizontal slider is created programmatically. It is an instance of the class HorizontalSlider located in the folder edu→uriship→math→parametric2d. The class is instantiated on line 105:

```
var hsSlider:HorizontalSlider=new
HorizontalSlider(this,3,30,435,250,"triangle");
```

The constructor takes parameters which control (in order of appearance): the target movie clip in which the slider will reside (in this case, the main movie referred to as "this", the depth in the clip, 3, the x and the y coordinates of the left endpoint of the slider, (30,435), the length of the slider,250, and the style of the knob. At this point there are two styles available: a triangle or a rectangle. The HorizontalSlider class has a number of methods which allow control of the appearance of an instance. Let's begin by changing "triangle" to "rectangle" in the constructor. Test your movie. The knob of the slider should now be rectangular.

Let's change the color of the knob to light blue. Scroll to line approximately 230:

```
hsSlider.changeKnobColor(0xCC0000);
```

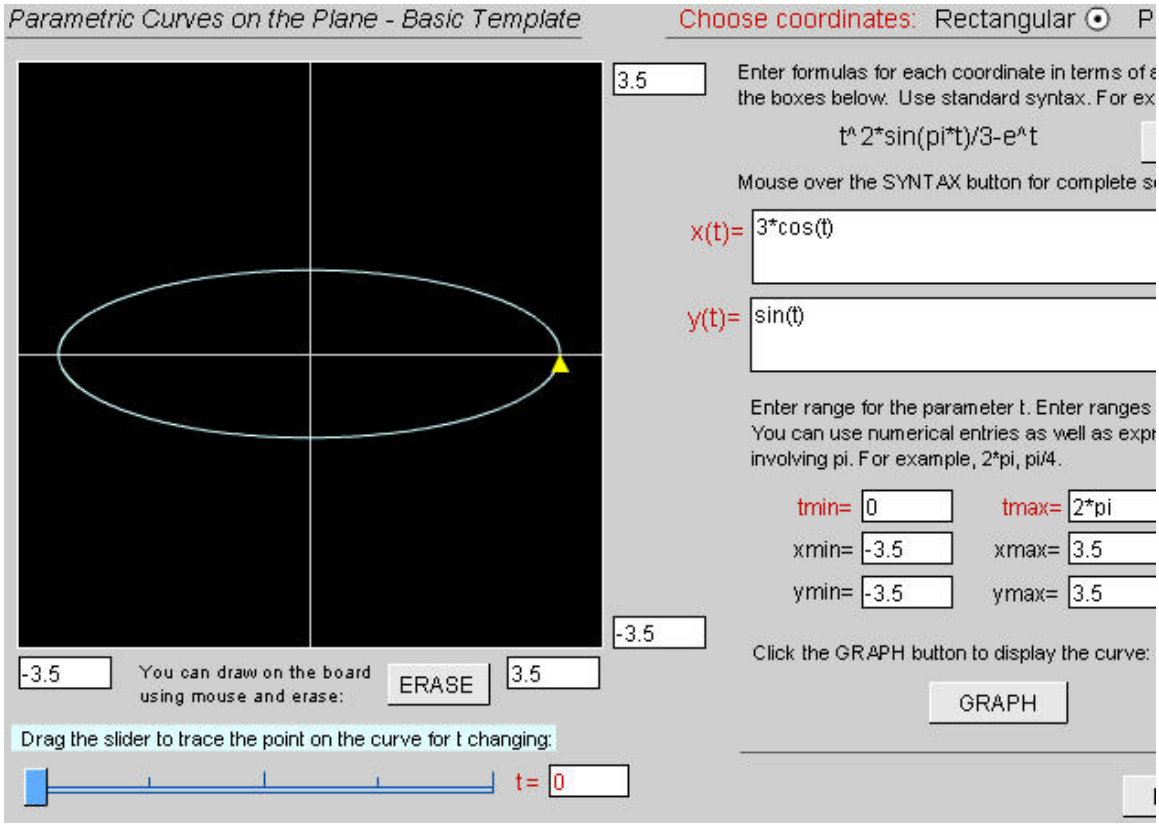
and change it to:

```
hsSlider.changeKnobColor(0x5AACFE);
```

You can also change colors of the track of the slider by evoking appropriate methods of the class. (We do not evoke them in the file, as the colors we use are default colors.) Let's change coloring of the track to blueish. Under the line changing the knob's color add:

```
hsSlider.changeTrackInColor(0xABD6FE);
hsSlider.changeTrackOutColor(0x0252A2);
```

The first line changes the color of the outline, the second the color of the inside of the slider's track. Test your movie. It should look as follows:



Picture 7

You have successfully changed the coloring of the grapher: pretty but there is room for improvement. See fg_guide.pdf accompanying our article *"Flash Tools for Developers: Function Grapher"* for tips how to change coloring of static text boxes and other elements on the Stage.

Modifying the Script: Resizing and Repositioning

If you want to change the layout of the grapher, the first thing you may want to change is the size of the graphing board and the size of the horizontal slider. This is done very easily by changes while calling the corresponding constructors. Go to line 62 again where "board" is instantiated:

```
var board:ParamGraphingBoard= new ParamGraphingBoard(20,20,350,this,1);
```

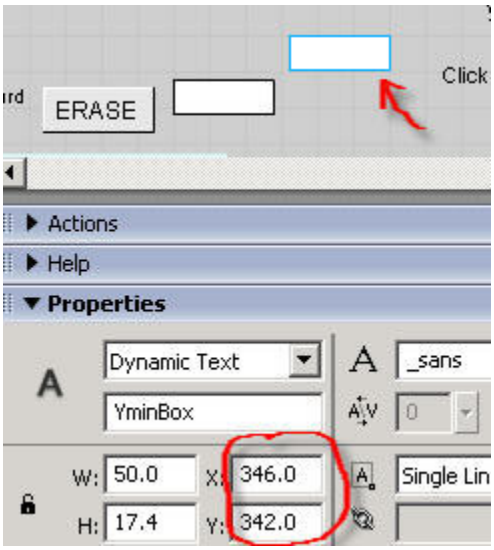
Change 350 to 250:

```
var board:ParamGraphingBoard= new ParamGraphingBoard(20,20,250,this,1);
```

Go to Control →Test Movie. As you see, the board is much smaller now but it has the same functionality as before. (If you forgot to delete the static box in blue font, it will stick out now from underneath the board. Delete it after you go back to the fla file. Afterwards, remember to click on "scripts" layer again so the Actions panel displays the

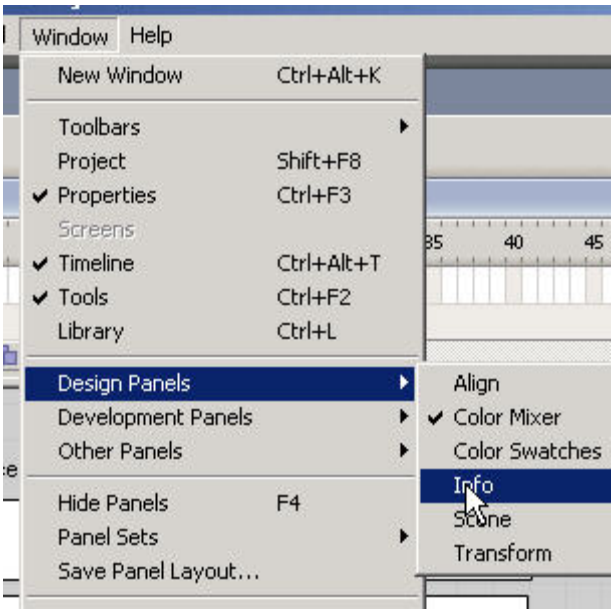
code.) After you resized the board, you may want to change the location of the range input boxes and their labels. You can do it manually in the fla file.

Close the Flash Player and go back to the fla file. Close the Actions panel and open the Properties panel. Select, for example, the ymin display box:



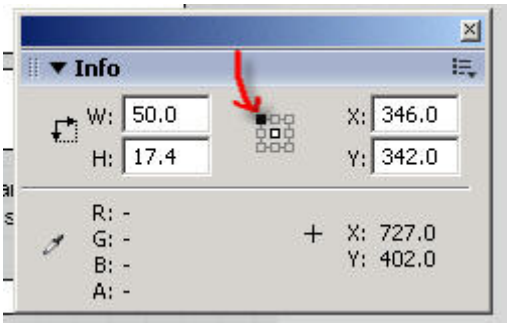
Picture 8

The Properties panel shows the box's instance name, YminBox, and its xy-position. You have decreased the size of the board by 70 pixels, so you want to move the box 70 pixels to the left and 70 pixels up. You can drag the box with the mouse, but you will achieve more precision (particularly not seeing the graphing board at the authoring time) by changing the box's coordinates in the Properties panel. Change the x-coordinate marked on the picture above to 276.0 and the y-coordinate to 272.0. Test the movie. The box is now in the right position. Similarly, you can change positions of other elements on the Stage. It is worth mentioning that the Properties panel will display the xy-coordinates of the left upper corner of a selected text box, which is what you want, provided you ensure such setting in the Info panel. To access the Info panel go to Window menu item, Design Panels, Info, and check it so the Info panel appears:



Picture 9

(In Flash 8, you access the Info panel directly from Window menu.) The Info panel appears:



Picture 10

Make sure the upper left rectangle is checked black as on Picture 10.

When resizing the graphing board, you usually have to adjust the positions and sizes of the error box and the coordinates display box. Test the movie and mouse over the graphing board. The coordinates display box is at an unsightly position, outside of the graphing area. Both the display box and the error box are parts of "board" and are controlled by it, but they do not have to be positioned within the graphing area. To change the positions of the boxes, go back to the fla file, open the Actions panel and scroll to line 363, approximately:

```
board.setCoordsBoxSizeAndPos(60,40,10,270);
```

The parameters control, respectively: width, height, x-coordinate, and y-coordinate of the box. The x- and the y-coordinates are relative to the graphing board; that is, relative to the upper left corner of the graphing board, which, in this example, resides at the point (20,20) of the main movie (see line 62). Change line 363 to:

```
board.setCoordsBoxSizeAndPos(60,40,10,200);
```

Test the movie. The coordinate box looks fine, but when you enter a formula with a syntax error the error box looks unsightly and sticks outside the board. Similarly as the coordinate box, the error box is a part of and is controlled by "board". Again, however, we can position it within or outside the graphing board. Let's leave it on top of the graphing board but adjust the error box's dimensions. Scroll to line 351:

```
board.setErrorBoxSizeAndPos(280,150,20,20);
```

The parameters control the width, the height, and the xy-position of the box relative to "board". Change the line to:

```
board.setErrorBoxSizeAndPos(210,150,20,20);
```

Test the movie and enter an erroneous formula. The error box looks fine now.

Modifying the Script: Changing Functionality

To change functionality of the graphers, we have to delve a little deeper into the code. In this little guide we will only suggest possible tutorials:

- Add preprogrammed examples of some spectacular looking curves. Create buttons named Example 1, Example 2, etc, and program their functionality so when the user clicks on them, preprogrammed curves are displayed and the corresponding entries appear in all boxes. In the guide fg_guide.pdf accompanying our previous article, we showed how to duplicate buttons, edit duplicates, and how to create buttons from scratch. Instead of creating buttons, you can create a so called blank button, a transparent button which you can place over a static text box. (See Ref. 1 or 2 for instructions how to do it.)
- Program Auto Range functionality where the applet chooses appropriate ranges for x and y so a curve entered by the user appears in its entirety. (You can find one way of doing that in param_advanced.fla.)

The Description of Custom Classes

The package edu.uriship.math.parametric2d contains seven custom classes: CompiledObject, MathParser, RangeObject, RangeParser, ParamGraphingBoard, HorizontalSlider, RangeParserTwo. The first four classes are identical as the classes with the same names contained in the package edu.uriship.math.fungraph which came with our article *"Flash Tools for Developers: Function Grapher"*. The last three are new. For the sake of completeness, we give below the description of all of the classes.

- 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 public 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 that 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(["t"]);
```

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 only one variable "t". There can be any number of variables, e.g.: new MathParser(["a","t"]), 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 has two public methods:

`procFun.doCompile(`*`string`*`)` -- this method takes a string (typically a string entered by the user) and returns an instance of CompiledObject. To give an example, we repeat below some of the code from param_basic.fla. The code resides within the drawCurve function.

```
var sFunction1:String="";

var compObj1:CompiledObject;

sFunction1=InputBox1.text;

compObj1=procFun.doCompile(sFunction1);
```

If `compObj1.errorStatus=1`, we know that the user has made a syntax error and we can send `comObj1.errorMes` to our error box for display. If `compObj1.errorStatus=0`, there is no error and we can send

```
compObj1.PolishArray
```

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

`procFun.doEval(`*`array, array`*`)` -- this method takes two arrays as parameters and returns a number. 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 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 is only one variable, "t", so the second array contains only one value, for example, tmin+tstep*i:

```
procFun.doEval(compObj1.PolishArray,[tmin+tstep*i]);
```

(The values tmin, tstep, and i are numerical variables which were defined earlier in the script, so tmin+tstep*i is a number representing a consecutive value for "t".)

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.

- RangeObject

This simple class defines a datatype that is returned by the parseRange method of RangeParser or RangeParserTwo. It is very similar as 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 or RangeParserTwo.

- 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. Here is an example of how we use it in our templates. The code appears within the drawCurve function.

```
var oRange:RangeObject;

oRange=procRange.parseRange(XminBox.text,XmaxBox.text,
                                        YminBox.text,YmaxBox.text);
```

If `oRange.errorStatus=1`, we send `oRange.errorMes` to our error box for display. If `oRange.errorStatus=0`, we use the `oRange.Values` array to set our range values:

```
xmin=oRange.Values[0];

xmax=oRange.Values[1];

ymin=oRange.Values[2];

ymax=oRange.Values[3];
```

- RangeParserTwo

The RangerParserTwo is a simple utility for parsing the user's input in the range boxes when the range for one variable is being defined. The class is almost identical to RangeParser except that its parseRange method takes two and not four parameters. RangerParserTwo allows 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 procRangeTwo:RangeParserTwo = new RangeParserTwo();
```

Each instance has only one method, parseRange:

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

The method takes two strings as parameters and returns an instance of RangeObject. Here is an example of how we use it in our templates. The code appears within the drawCurve function.

```
var oRange:RangeObject;

var sTmin:String;

var sTmax:String;

sTmin=TminBox.text;

sTmax=TmaxBox.text;

oRange=procRangeTwo.parseRange(sTmin,sTmax);
```

If `oRange.errorStatus=1`, we send `oRange.errorMes` to our error box for display. If `oRange.errorStatus=0`, we use the `oRange.Values` array to set our range values:

```
tmin=oRange.Values[0];

tmax=oRange.Values[1];
```

- ParamGraphingBoard

This class is responsible for creating all visual elements of our graphers except for those that were created at the authoring time. It is very similar to the class GraphingBoard in edu.uriship.math.fungraph package. Its constructor:

```
var board:ParamGraphingBoard=new ParamGraphingBoard(20,20,350,this,1);
```

and its methods are used and commented exhaustively in the templates' scripts as well as in the sections above. Below we list the public instance methods of the class with examples of possible parameters (if they take parameters). We are assuming that the instance is stored in a variable called "board" as in our templates. The methods below return nothing except for `board.drawGraph` discussed below.

```
board.changeBorderColor(0xFFFFFF)
board.changeBackColor(0x000000)
board.enableErrorBox()
board.disableErrorBox()
board.setErrorBoxFormat(0x000000,0x000000,0xCCCCCC,12)
board.setErrorBoxSizeAndPos(300,150,20,20)
board.enableCoordsDisplay()
board.disableCoordsDisplay()
board.setCoordsBoxFormat(0x000000,0x000000,0xCCCCCC,12)
board.setCoordsBoxSizeAndPos(60,40,20,300)
board.enableUserDraw(0xFFFF00,1)
board.disableUserDraw()
board.setAxesColor(0xCCCCCC)
board.setVarsRanges(-10,10,-10,10)
board.drawAxes()
board.drawGraph(1,fArray,0xFF0000)
board.eraseUserDraw()
board.eraseGraphs()
```

The latter method erases functions' graphs as well as deletes range settings. It does not erase the user's sketches. It should be noted that the method `board.setVarsRanges` should be called before `board.drawAxes` or `board.drawGraph` are called. We do so in all templates, for example, within the drawCurve function:

```
board.setVarsRanges(xmim,xmax,ymin,ymax)
```

after xmin, xmax, ymin, ymax are already defined. It should be also noted that the method `board.drawGraph(1,fArray,0xFF0000)` besides drawing curves, returns an array which records consecutive positions and rotations of the tracing arrow along the curve being drawn. This fact is used in drawCurve function:

```
locArrowPos=board.drawGraph(1,fArray,0xFF0000);
```

Here are methods related to tracing functionality and the appearance of the arrow. They return nothing.

```
board.enableTrace(true)
board.arrowVisible(true)
board.setArrowPos(a,b,c)
```
(Works if a,b are xy-coordinates in pixels relative to "board", c is the rotation)

```
board.changeArrowColor(0xCC0000)
```

The following methods are also public and may be useful at times:

`board.xtoPix(a)` (Converts from functional value "a" for x to pixels after xy-ranges are set. Returns a number.)

`board.xtoFun(a)` (Converts from pixel value "a" for x to a functional value. Returns a number.)

Similarly:

```
board.ytoPix(a)
board.ytoFun(a)
```

The following methods might be useful at times:

```
board.getBoardSize()
```

which return the size of the graphing board. Also:

```
board.isDrawable(a)
```

A given entry "a" usually represents one of the coordinates in pixels of a point to be drawn or the cursor to be moved to. The method checks if "a" is a finite number and if it does not exceed the pixel value beyond which the undesirable effect of "wrapping around" may occur. (This value is set at 5000 pixels). It returns "true" or "false".

The only instance method of ParamGraphingBoard class never mentioned or used is

```
board.destroy();
```

The method removes listeners, deletes movie clips created by board, etc. You should evoke the method before deleting the variable "board" (should you need to delete it.)

- HorizontalSlider

This class is responsible for creating a horizontal slider of desired length and appearance. Its constructor:

```
var hsSlider:HorizontalSlider=new
HorizontalSlider(this,3,30,435,250,"triangle");
```

and its methods are used and commented exhaustively in the templates' scripts as well as in the sections above. Below we list the public instance methods of the class with examples of possible parameters (if they take parameters). We are assuming that the instance is stored in a variable called "hsSlider" as in our templates. The methods below return nothing.

```
hsSlider.changeKnobColor(0xCC0000)
hsSlider.changeTrackOutColor(0x0000FF)
hsSlider.changeTrackInColor(0x0000FF)
hsSlider.changeKnobRightLine(0x000000)
hsSlider.changeKnobLeftLine(0xFFFFFF)
hsSlider.setKnobPos(a)   (Sets the position of the knob on the slider
                          "a" pixels from the left endpoint.)
```

The following methods may be useful at times:

```
hsSlider.getSliderLen() (Returns slider's length.)
hsSlider.getKnobPos()   (Returns knob's position in pixels relative to the left
                         endpoint.)
hsSlider.getSliderLen()
hsSlider.destroy()
```

The latter method removes listeners, movie clips, etc. created by an instance of HorizontalSlider. You should call the method before deleting a variable, like hsSlider, holding an instance.

If you have any doubts about any of the methods for any of the classes in the package, navigate to edu→uriship→math→parametric2d and open the corresponding .as file in Flash. You will see the code for each class.

Final Remarks

The movie clip "syntax" that appears when a user mouses over the SYNTAX button does not reside on the Stage. It is attached at runtime (line 122):

```
var mcSyntax:MovieClip=this.attachMovie("syntax","syntax1",10);
```

For the latter command to work, there must be linkage established from the movie clip "syntax" which resides in the Library to ActionScript. Our previous guide fg_guide.pdf explains the process of linking in detail.

If you want to edit the "syntax" clip, drag it on the Stage, preferably to the empty layer "scripts", right-click on it, choose Edit in Place. After editing, go to Edit→Edit Document and delete the clip from the Stage.

The radio buttons for choosing a coordinate system are movie clips which have been created by hand and not radio buttons components available in Flash. If you want to see how the movie clip is created, right-click on it and choose Edit in Place. You will access the timeline for the radio clip.

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 Macromedia site:

http://www.macromedia.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 for Windows and Macintosh: Visual QuickStart Guide, Peachpit Press, 2004. Flash 8 version is due in December 2005.

2. Robert Reinhardt and Snow Dowd, Macromedia Flash MX 2004 Bible, Wiley Publishing, 2004. Flash 8 version is due in February 2006.

To learn ActionScript 2.0 (which hasn't changed much between Flash MX 2004 and Flash 8):

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

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

5. 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.