Modeling Science: From Free Fall to Chaos

Wolfgang Christian and
Francisco Esquembre

PART 1

Modeling With *Easy Java Simulations*

Chapter Three

EJS and Java Concepts

> A complex system that works is invariably found to have evolved from a simple system that works. *John Gaule*

In order to study the relation between *EJS* and Java, we build time independent and time dependent function plotters from scratch. We introduce common data types including *objects* and we explore various user interface elements and show how to control their appearance. Finally, we show how *EJS* makes it easy to display scalar and vector fields.

3.1 DESIGNING A FUNCTION PLOTTER

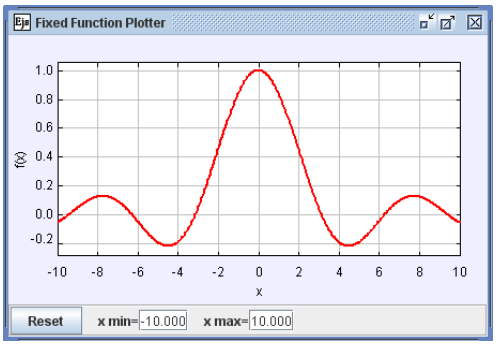


Figure 3.1: The Fixed Function Plotter user interface. The frame (window) contains a plotting panel with axes and a $\sin x/x$ plot and a control panel with a button and input fields.

Although it is not necessary to become a Java expert to use *EJS*, a basic understanding of Java programming concepts and how *EJS* implements these concepts will help us write clearer, more efficient, and more attractive models. As in Chapter 2, we again use simple models to illustrate, but this time we build our models from the ground up to show the details of how they are constructed. Our first example displays a function $y = f(x)$. Because we have a good idea what such a model should look like, we start by

building the view (user interface) shown in Figure 3.1, and then define the variables and methods needed to create the model's data. We use this basic view in subsequent sections to display functions. Along the way, we study Java syntax, data types, objects, and graphical user interfaces.

Start *EJS*, press the new simulation icon on the taskbar to clear all workpanels, and select the View workpanel. Save the model in your *EJS* user directory using a name such as `FixedFunctionPlotter`. Select the *Windows, containers, and drawing panels* tab on the interface palette and create a frame by selecting the frame icon on the palette and then clicking on the Simulation view in the tree of elements. A *frame* is a top-level user interface element with a title and a border which serves as a container for other elements. Right-click on the name of the newly created frame in the tree and note that this frame is tagged as the *Main window* of the simulation. The main window is special because a running simulation will exit (terminate) when its main window is closed. The first frame created is assumed to be the main window, although you can change the main window designation by setting this property in another frame.

A frame is useless without content and we now create a *plotting panel* with x- and y-coordinate axes by selecting the `PlottingPanel` icon on the palette and clicking on the frame in the tree of elements. Two dialogs appear asking for the name of the plotting panel and the position of the panel within the frame. Choose the center position so that the plot occupies this region of the frame. Double-click (or right-click) on the name of the newly created plotting panel in the tree to show its properties inspector. Set the Title property to `Function Plotter` and the Title X and Title Y properties to `x` and `f(x)`, respectively. (Recall that *EJS* will add quotes to strings entered, if needed.) The Autoscale Y property should be set to `true` because we do not know the range of y values. Set the Autoscale X property to `false` and enter `xmin` and `xmax` for the Minimum X and Maximum X properties, respectively.

Although we have entered characters on a keyboard, *EJS* has made a number of (reasonable) assumptions about what we have typed into the inspector. It assumes that title entries are literal and should not be interpreted. Behind the scene, *EJS* converts each title property into a Java `String` variable and uses it to set the corresponding title. The autoscale entries `true` and `false` are interpreted as Java `boolean` variables. Again, conversion is done behind the scene, and the boolean value is used to set the behavior of the plot. The `xmin` and `xmax` property values are assumed to be entries in a Variables table of the Model workpanel. Because these variables have not yet been defined, their property fields have a pink background. Go to the Variables section of the Model workpanel and define `xmin` and

`xmax` variables with initial values −10 and 10, respectively, as described in Section 2.6. Return to the inspector and observe that the field backgrounds are now white and the value of the model variables has been applied to the plotting panel's maximum and minimum.

A plotting panel, as seen within the frame in Figure 3.1, has titles and axes but it does not store or draw xy-data points. Navigate to the 2D drawables palette and select the trace icon from the collection of basic elements. When the wand cursor appears (see Figure 2.14), click on the plotting panel in the tree of elements to add a *Trace* element to the plot.

The trace stores data points and draws them within a plotting panel. Show the trace inspector and enter `x` and `y` for the Input X and Input Y properties, respectively. The input field backgrounds will again turn pink because these variables have not yet been defined. Navigate to the Variables panel and create an integer variable `n` with a value of 500 and double variables `x` and `y`. Be sure to change the data type for `n` from `double` to `int` and do not include a decimal point when entering the value. Integer data is stored and processed differently from real (floating point) numbers and the value field will signal an input error by turning pink if you include a decimal point when entering an integer value. Because we want the `x` and `y` variables to represent arrays of numbers, specify their dimension (number of array elements) by entering `[n]` in the Dimension column. Figure 3.2 shows the completed table for the Fixed Function Plotter.

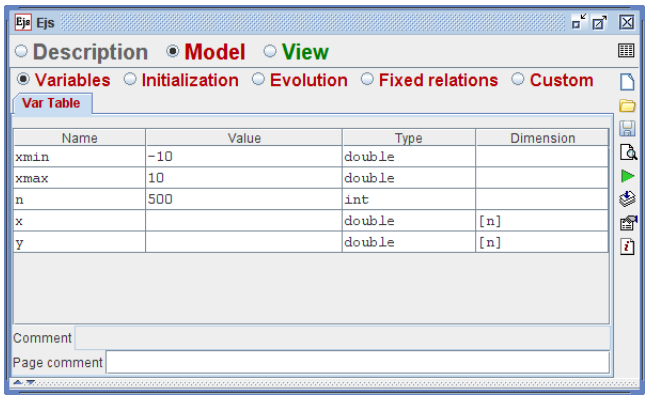


Figure 3.2: The Fixed Function Plotter variables table.

The *Dimension* column in a variable table allows us to incorporate vectors and matrices. The technical term for this type of variable is an *array*. Arrays can have one or more dimensions and each dimension is specified by the number of array elements in this dimension surrounded by square brackets.

Java containers, such as frames, use a *layout manager* to position and size graphical user interface components and the most commonly used managers are `BorderLayout`, `FlowLayout`, and `GridLayout`. The `BorderLayout` is the default layout manager for a frame. Adding a plotting panel to the main frame showed us that the available positions for a border layout are `Center`, `Up`, `Down`, `Left`, and `Right`. The `FlowLayout` and `GridLayout` layout managers determine component location based on the order in which components are added to the container.

Figure 3.3: A layout manager determines the position of an object in a container. Available positions for the border layout are `Center`, `Up`, `Down`, `Left`, and `Right`.

A *panel* is a simple (lightweight) container without decoration, such as a title, that is used to group components such as buttons and input fields. Select the panel icon ▭ on the interface palette and click on the frame to create the element. Name this panel `controlPanel` and place it in the down position of the frame. Because we want to group our user interface components near the bottom of the frame, create the panel in the frame's down position. Double click on the newly created panel's name in the tree of elements and set the Layout property to `FLOW` and the Border Type property to `LOWERED_ETCHED`. You can type these values into the property fields, but it is much easier to click on the editor icon (the first icon) to the right of each variable field to make these assignments. Resting the mouse on an icon in the Border Type editor shows a hint with the name of the border.

The control panel is almost invisible in the function plotter mock up (design) because the panel has no content and its default size is zero except for the border. Select the *Buttons and decoration* tab on the interface palette and add a button named `resetButton` to the control panel by selecting the button icon on the palette and then clicking on the control panel node in the tree of elements. Set the Text property of this element to `Reset` and the Action property to `_reset()`. The button's action is a predefined *EJS* method (function) that will be invoked when the button is pressed. Click on the action list (second) icon to see predefined actions such as those shown in Table 3.1. Because internal *EJS* methods and variables begin with the underscore character, users should not define methods or variables that begin with this special character.

We now create a panel named `xMinPanel` nested inside the control

Table 3.1: Examples of predefined *EJS* methods than can be used throughout a model. A complete list is available in Appendix XX and in the *EJS* help.

`_initialize()`	Executes code on the Initialize panel and executes code on the Fixed relations panel
`_isPaused()`	Returns `true` if the simulation is not running.
`_pause()`	Pauses (stops) the simulation.
`_play()`	Plays (starts) the simulation.
`_reset()`	Resets the simulation to its initial state.
`_view.resetTraces()`	Clears data from all traces in the simulation.

panel by clicking on the panel icon ▭ and then clicking on the control panel node in the tree of elements. This panel will layout a label and a number field as shown in Figure 3.4. Create the label in the panel's left position and set its text property to `x min =`. Locate the Field element on the input and output palette and create it in the panel's center position. Set the number field's Format property to `0.00` and set the Variable property to `xmin`.

Create a second group of elements for the maximum x value by making a copy of the x-minimum panel. Right-click on the name of the x-minimum panel in the tree of elements and select Copy and then right-click on the control panel name and select Paste. Because element names must be unique, *EJS* automatically renames the newly created elements to avoid conflicts. You should rename the new elements by right-clicking on the nodes in the tree so that the names convey meaning. For example, change the copied panel's name to `xMaxPanel` so that the user interface appears as in Figure 3.4. Use the element inspectors to set the Text property of the copied label to `x max =` and the Variable property of the copied number field to `xmax`. You can also set an element's Size property to suggest a preferred size if the default is inappropriate.

Adding components or changing element properties causes the container's layout manager to arrange and resize the elements. You can right-click on elements in the tree of elements to rearrange their order.

Exercise 3.1. Layout manager
Use the properties inspector to change the control panel's layout manager from a flow layout to a grid layout with a single row and three columns.

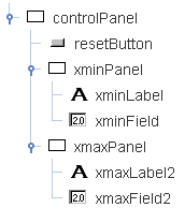


Figure 3.4: Nested panels are used to group user interface elements.

Resize the frame and observe how the user interface changes appearance. Is the flow layout or the grid layout more appealing? □

3.2 FIXED FUNCTIONS

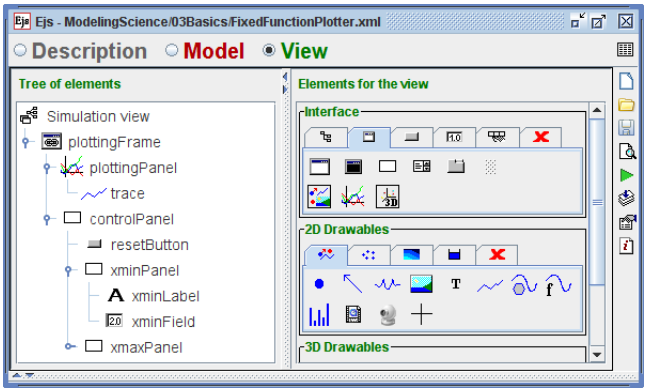


Figure 3.5: The tree of elements shows the structure of the Fixed Function Plotter user interface.

The user interface is now complete, and we have not had to write any low-level Java code. *EJS* creates the code for the frame, button, and number fields when the run button is pressed on the *EJS* taskbar.

We are now ready to provide code for the model. In order to illustrating programming concepts, we implement our function by coding it as a Fixed relation of the model. As explained in Section 2.3.2.4, fixed relations are evaluated when the model is initialized and whenever the model changes. Changing the `xmin` or `xmax` values by editing their input fields will therefore recompute the function.

Listing 3.1: A fixed relation is evaluated whenever the model changes.

```
double xLocal=xmin;                    // independent variable
double dx=(xmax-xmin)/(n-1);           // variable increment
_view.resetTraces();                   // clear old data from trace
for(int i=0; i<n; i++){                // set up loop
```

```
  x[i] = xLocal;                          // store independent variable in array
  if(xLocal==0) y[i] = 1;                 // function is one if xLocal=0
    else y[i] = Math.sin(xLocal)/xLocal; // otherwise, evaluate function
  xLocal += dx;                           // increment independent variable
}                                         // end of loop body
```

Create a fixed relation code page for the function plotter and enter the fixed relation code in Listing 3.1. Note the following options. You can change the code editor's font size by selecting the Edit options icon on the taskbar and selecting the Change font option on the Aspect tab. Words native to the Java language, such as `double` and `for`, are reserved and are shown using a blue font. Java reserves fifty four words so the language itself is very compact. Learning what can be and has been done with the language is the difficult part. A double slash begins an end of line comment and these comments are shown using a red font.

Java code is organized into classes (files) and the `Math` class contains the code needed to perform common mathematical operations. If you type the name of the mathematics class `Math` followed by a dot, the editor displays a list of suggested completions including constants such as `PI` and functions such as `sqrt`. Functions in the mathematics class are displayed with their arguments and the variable type that is returned. The `sqrt` function for example, takes a `double` input parameter and returns a `double` value with its square root. The `+=` operator is shorthand for a combination of an addition followed by an assignment `xLocal = xLocal + dx`. The Java syntax and the standard libraries are described more fully in Appendix B.

There are many ways to code a model and there are tradeoffs between clarity, roundoff error, and efficiency. For example, every iteration of the loop in the fixed relation code accumulates roundoff error when the independent variable is incremented `xLocal += dx` because the value of `dx` is only stored with 16 significant digits. Computing the independent variable within the loop body using (`double xLocal = xmin + i*dx`) reduces the roundoff error but is less efficient because of the added multiplication. These considerations are not important in our simple model because the accumulated error and the change in performance are small. A good rule of thumb when building a model is to favor clarity and to adjust the model in order to improve performance after the model has been debugged and its correctness has been verified.

The fixed relation code in Listing 3.1 illustrates many important programming concepts. The first statement defines a new `double` variable `xLocal` that will be incremented from `xmin` to `xmax`. The value of `xLocal` is set to the initial x-value `xmin`. The second statement defines the increment `dx` between data points and computes its value. The third statement invokes

a predefined method from Table 3.1 to clear old data. The fourth statement starts a loop with a body containing three statements. A loop typically requires the declaration and initialization of a counter variable (`int i=0`), a test to determine if the counter variable has reached its terminal value (`i<n`), and a rule for changing the counter variable (`i++`) after every iteration. These three parts of the `for` loop are contained within parentheses and are separated by semicolons.

```
 // for loop template
for(initialization; continue-condition; end-of-loop-operation);
```

Note our use of the increment operator `++` for our end of loop operation rather than the equivalent expression `i = i + 1`. The body of our loop is executed `n` times and each iteration assigns a value to successive array elements `x[i]` and `y[i]`.

It is customary to indent statements within a code block, such as the body of a loop, so that it can be easily identified. Java ignores indentation (whitespace), but it provides an important visual cue to the code's logical structure. You can reformat a code page to indent code by right-clicking within the page and selecting the format option.

Where you define variables is important. The `x`, `y`, `xmin`, `xmax`, and `n` variables are defined in the model's Variables table (see Figure 3.2) and can therefore be used throughout the model. These variables are *global*. The `xLocal`, `dx`, and `i` variables are defined in the fixed relation code page and can only be used within that page. These variables are *local*. It is considered bad programming to clutter up a model with unnecessary global variables. It is far better to define and use local variables on the code page where they are needed. A short end of line comment is all that is needed to document a local variable's purpose. Note that if we define a local variable using a global variable's name, for example, if we had written the first statement as `double x=xmin` rather than `double xLocal=xmin`, then the local variable hides (shadows) the global variable and the global `x` variable becomes inaccessible.

The Fixed Function model is now complete. Run the model to insure that you have not made syntax errors.

Exercise 3.2. Adding a trace
Add a second trace to the model to show the $\sin x$ function as well as the $\sin x/x$ function. □

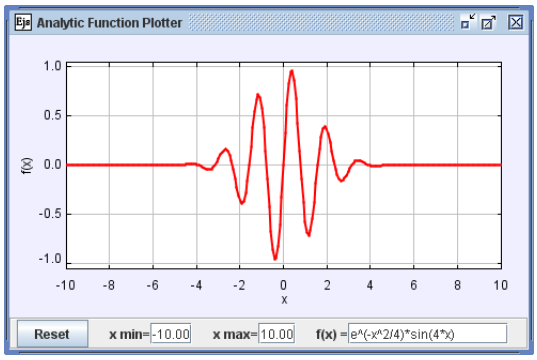


Figure 3.6: The Analytic Function Plotter uses a *parser* to convert keyboard input into a mathematical expression. The function $e^{-x^2/4} \sin 4x$ is shown.

3.3 INPUTTING FUNCTIONS

The Fixed Function Plotter is useful if the function we wish to display is known to the modeler, but changing functions requires that we recompile the model. In order to build models that allow users to input functions, we need to convert a sequence of characters into a mathematical expression. Code that does this conversion is known as a *parser*. The *Analytic Curve* element uses a parser to plot a function that is represented as a string. The string can contain any of the functions shown in Table 3.2 and can be changed at any time.

Table 3.2: The *EJS* mathematical expression parser accepts commonly used one- and two-parameter functions.

abs(a)	acos(a)	acosh(a)	asin(a)	asinh(a)
atan(a)	atanh(a)	ceil(a)	cos(a)	cosh(a)
exp(a)	frac(a)	floor(a)	int(a)	log(a)
random(a)	round(a)	sign(a)	sin(a)	sinh(a)
sqr(a)	sqrt(a)	step(a)	tan(a)	tanh(a)
atan2(a,b)	max(a,b)	min(a,b)	mod(a,b)	

Start with the Fixed Function Plotter model and save it as **AnalyticFunctionPlotter**. Navigate to the variables table and remove the number of elements variable `n` and the array variables `x` and `y` by right-clicking on the rows in the table. Add a string variable named `functionString` to the table to store the function and set its value to `"sin(x)/x"`. The quotes around the value tell *EJS* that we are entering a string and that the entry is literal. Navigate to the Fixed relations workpanel and right-click on the panel tab to remove this code page. Navigate to the View workpanel and right-click on the trace in the tree of elements to remove it. Add an `Analytic Curve`

element f from the basic 2D drawable palette to the plotting panel.

Add a third panel named `functionPanel` to the `controlPanel`. This new panel will be similar in appearance to `xMaxPanel` except that we use a text field rather than a number field to interpret user input as a string rather than as a number. Name the text field `functionField` and set its width and height Size property to `150,20` pixels. Set its Variable property to `functionString` by clicking on the field's link icon and selecting that variable. The variable name appears in the field surrounded by percent signs `%functionString%` as shown in Figure 3.7 because the input is a variable name. Input not surrounded by percent signs is assumed to be the mathematical expression that we wish to evaluate. This ambiguity between a literal expression and a variable name only occurs when a property value is a string.

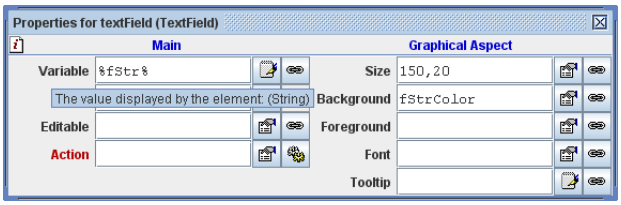


Figure 3.7: Character input can be ambiguous because a string can be interpreted as a literal expression or a variable name. Percent signs surrounding a string alert *EJS* that input, such as `%functionString%`, is a variable. Note that the property data type is displayed within the tool tip that is shown when the mouse hovers over the property name.

> Although the Java language requires that strings be defined within quotation marks, *EJS* relaxes this requirement when text is used in an inspector. For example, the Text property in a button inspector can be set using either `"Reset"` or `Reset`. *EJS* will automatically add the necessary quotes. Surrounding a property value with % forces *EJS* to look for the corresponding variable in the model. The use of % characters is optional for properties which don't accept strings as input as there is no possible ambiguity. *EJS* property inspectors only require us to use percent characters % to distinguish between a string and a variable name when the property that is being set is itself a string. Resting the mouse on a property name will show the property data type as seen in Figure 3.7.

Set the Background property of the `functionField` to `functionFieldColor` so that we can control its background color and navigate to the Variables table to create the `functionFieldColor` variable. A color variable consists of three integers in the range $[0, 255]$ that determine a mixture of primary colors. For example, yellow is a mixture of red and green and can be stored as $(255, 255, 0)$ where the integers represent red, green, and blue. Compound

variables, such as color, require more memory, and are referred to as *objects* so we choose `Object` as the `functionFieldColor` variable type.

Variables written with all capital letters are usually fixed (immutable) constants that can be used throughout a program. The mathematics class, for example, defines the `Math.PI` and `Math.E` constants. Because certain colors are frequently used, Java defines color constants such as `java.awt.Color.RED` and `java.awt.Color.YELLOW`. Java programmers capitalize variable names that cannot be changed so that it is easy to recognize `PI` and `YELLOW` as constants.

Long names are a convenient way of organizing variables and code to avoid naming conflicts. We could, for example specify the creator of *EJS* by writing `earth.europe.spain.murcia.Esquembre.FRANCISCO` to distinguish him from any other Francisco. Long names are, however, awkward and although we can enter `java.awt.Color.WHITE` as the value for the `functionFieldColor` variable, we can shorten the name by importing the class where `Color` is defined. Open the simulation information panel [1] and enter the `Color` class into the import statement list by clicking on the add icon ✓ and typing the full class name `java.awt.Color`. Java can now find the `Color` class and we can refer to color constants using shorter names such as `Color.WHITE`.

We are now ready to set the analytic curve's properties in the View. The Variable property should be set to `x` because it specifies the independent parameter that will be incremented. The X() and Y() properties are functions of this independent parameter and are strings that will be processed by a parser. The x-coordinate function is a string (literal) that does not change so we enter it as `"x"`. The y-coordinate function is a variable that does change so we enter the variable name surrounded with percent signs as explained previously. The Java Syntax property allows us to parse code expressions such as `Math.sin`, but it is set to `false` because we wish to input functions using standard mathematical notation. We could specify the number of points using the Points property, but this is unnecessary as *EJS* automatically chooses a sufficient number of points to give a smooth curve. Minimum and Maximum properties are also not specified because we are plotting a function of x, and the analytic curve uses the x-axis minimum and maximum values as defaults.

Users often make input errors when typing complicated expressions. In order to give modelers the opportunity to respond to these input errors,

[1] The information panel contains textual data about the simulation provided by the author, as well as instructions to *EJS* that affect the three parts of a simulation, such as additional files or import statements. Click on the top right icon of *EJS* ▦ to display the simulation panel.

the analytic curve implements On Parse Error and On Parse Success actions. The Analytic Curve Plotter uses these actions to set the value of the `functionFieldColor` variable. Enter `functionFieldColor=Color.RED` for the On Parse Error action and enter `functionFieldColor=Color.WHITE` for the On Parse Success action. Because the `functionFieldColor` variable is bound (connected) to the `functionField`, the text field's background color will alert the user if she has typed an incorrect expression. Run the Analytic Function Plotter to see that it performs properly with a wide range of functions. Note again that this simulation was created without writing very much Java code.

Exercise 3.3. Mathematical expression parser
Predict the shape of the curve produced by the following expressions and test your prediction using the Analytic Function Plotter.

- `int(x)`
- `floor(x)`
- `sign(sin(x))`
- `mod(x,1)`
- `frac(2*sin(x))`
- `max(0,sin(x))`

Explain what each of the functions, such as `mod`, do. □

3.4 OBJECTS

You have probably noticed that we refer to plots, panels, and parsers as tangible things (objects) with properties such as title, size and color. This terminology is no accident. When we design a view by selecting *EJS* elements from a palette, we are creating Java *objects* in computer memory. The "black magic" required to create Java objects is, in fact, very simple. Behind the scene, *EJS* uses the `new` operator to construct (instantiate) whatever is requested. For example, selecting the trace on the basic 2D drawables palette and clicking twice with the magic wand on a plotting panel within the tree of elements creates Trace elements such as `traceOne` and `traceTwo`. This action produces (hidden) code similar to the following:

```
// Do not use.  Pseudocode will not put traces into view.
Trace traceOne = new Trace();  // create traceOne variable and construct Trace object
Trace traceTwo = new Trace();  // create traceTwo variable and construct Trace object
```

Each trace is made (instantiated) using the same blueprint (constructor), and each stores its own line color and its own xy-data and each has a unique name that identifies the object. *EJS* element names, such as `traceOne` and `traceTwo`, identify variables just as variable table entries identify variables. Variables of type `double`, `int`, and `boolean` are *primitive variables* because they have only a value property. Object variables are more powerful because they contain multiple primitive variables. Objects must be instantiated with the `new` operator before they can be used.

An object is more than just a collection of primitive variables. Java objects contain methods (functions) that can be applied to their data. We can, for example, change the line color and add data to trace objects using the `setColor` and `addPoint` methods, respectively, as shown in the following code fragment:

```
traceOne.setColor(Color.RED);   // set the color
traceTwo.setColor(Color.BLUE);
traceOne.addPoint(0,3);         // add (x,y) data
traceTwo.addPoint(1,4);
```

The use of a "dot" to access a variable or to invoke a method is common notation, and we have already used this syntax to access trigonometric functions in the `Math` class and colors in the `Color` class. Invoking methods directly is low-level programming and *EJS* is designed to minimize such programming by letting us use inspectors. For example, entering `x` and `y` variable names as X and Y properties in the trace inspector (see **FixedFunctionPlotter**) causes *EJS* to create code that invokes the `addPoint` method behind the scene. There are, however, situations when we wish to directly invoke an object's method and we do so in the next example.

> Although the coding of objects is a job for Java programmers and not the goal of this book, readers may be interested to know that Java programmers often build new objects from existing objects. The plotting panel, for example, is built by adding axes, color, and scale properties to a standard Java panel. This is convenient because it saves effort. Sun Microsystems, the creator of Java, programmed the panel for us and all that we, the developers of *OSP* and *EJS*, needed to do was to add the code needed to produce the plot. The ability to define new objects by adding to existing objects is known as *inheritance*.

To convey a sense of time development, we plot a function $y = f(t)$ in real time by generating data at a constant (fixed) rate as shown in Figure 3.8. Create a new model named **TimeDependentFunctionPlotter** and save it in your working directory. The user interface will be similar to our previous

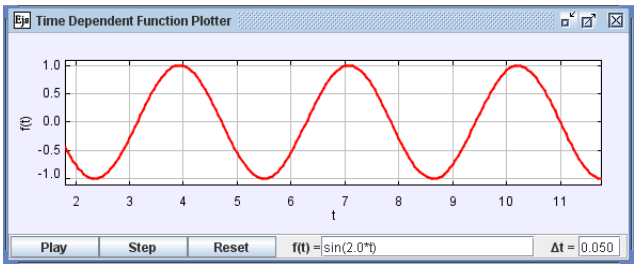


Figure 3.8: The Time Dependent Function Plotter generates and plots data at a constant rate.

function plotters. Create a main window with a plotting panel and a control panel using the appropriate *EJS* elements or open a previous example in a second instance (copy) of *EJS* and copy and paste from that view. Set the plotting panel's titles and add a trace named `trace` to the panel. Set the plotting panel's Autoscale X property to `true` and its Maximum X property to `100*dt`. Because both properties are set, the x-axis will only autoscale if the trace data has a value larger than the Maximum X property. The x-axis minimum will always autoscale because the Minimum X property has not been set. The y-axis minimum and maximum will autoscale to exactly fit the data, but set the Y Margin property to 10 so that the scale is increased by 10% to leave extra space around the plot. Use the trace inspector to set the Max Points property to 200, the Line Color property to `RED`, and the Stroke property to 2.

Add a panel to the control panel named `buttonPanel` and create a two-state button within this panel using the icon on the buttons and decoration palette. Inspect the two-state button and note that a number of its properties are already set. Create a regular (one-state) button within the button panel named `stepButton`, set its Text property to `Step`, and set its Action to `_step()`. Create another button within the button panel named `resetButton`, set its Text property to `Reset`, and set its Action to `_reset()`. Set the button panel's layout manager to grid layout and specify one row and three columns.

Add a panel named `functionPanel` to the control panel in order to group together a label and a *Function* element f(x). Create the label in the panel's left position and set its text property to `f(t) =`. A *Function* element is a text field that passes its input directly into a parser. If the user enters an incorrect mathematical expression, the field automatically turns red. Locate the Function element on the *input and output* palette and create an instance named `analyticFunction` in the panel's center position. Set the Function property to `%functionString%` and set the Action property to `_initialize()`. This action executes the code on the Initialization panel

followed, as usual, by the code on the Fixed relations workpanel. The reset button action does this same initialization after it clears data from all view elements and sets variable values using the entries in the Variables workpanel.

Add a third panel named `dtPanel` to the control panel to group a label and number field. Create the label and set its Text property to `$\Delta$t =` and note that the mock up shows Δt. *EJS* supports common Greek characters within text strings by surrounding the name of the letter with `$` characters. For example, the α, β, and θ characters are `\alpha`, `\beta`, and `\theta`. The capital Greek character is displayed if the first letter in the name is upper case. Create a field named `dtField`, set its Variable property to `dt`, and set its Action property to `_initialize()`.

Property fields in the `analyticFunction` and `dtField` inspectors are pink because neither the `functionString` nor the `dt` variables are defined. Navigate to the Variables workpanel and add `t`, `dt`, and `functionString` variables. Set the `dt` value to 0.05 and set the `functionString` value to `"sin(2.0*t)"`. Note that string variables must be enclosed within quotation marks within the Variables workpanel. The user interface is now complete.

The two-state button's default properties are set to control a model that evolves in time. The On Action property invokes the `_play()` method and the Off Action property `_pause()` method. Navigate to the Model workpanel, select Evolution, and then click within the upper half of the panel to create an explicit evolution page. Name this code page `Time Step` and enter the following code:

```
t += dt;                                          // advance time
double u = _view.analyticFunction.evaluate(t);  // evaluate function
_view.trace.addPoint(t,u);                        // add point to trace
```

The first statement evolves the model by adding the time step Δt to the current value of time t. The second statement evaluates the analytic function at time t and stores the value in the variable u. The third statement adds a data point to the trace using the global variable `t` and the local variable `u` as the xy-data point.

There are many objects within an *EJS* model and *EJS* organizes them based on where they are used. Because the analytic function and trace are created and used in the View, the complete variable names are `_view.analyticFunction` and `_view.trace`. Appending a method name to a variable name allows us to perform (execute) the corresponding method.

> Object-oriented programming is based on the assumption that a modeler must know what an object (piece of code) does but need not know how it does it. This hiding of code is know as *encapsulation*. Whenever an object does something, we say that the object *performs* a method or that we *invoke* a method. A partial list of methods can be obtained by clicking on the small information icon in the upper left hand corner of the element's inspector. See the *OSP Guide* [**?**] for a more complete description of objects within the *OSP* library.

The frames per second (FPS) field on the Evolution panel specifies how many times per second the simulation repaints the screen. The steps per display (SPD) field determines how many evolution steps are performed between each repaint operation. Data will be generated in real time because the FPS field is 20, the SPD field is 1, and the time step Δt is 0.05.

Navigate to the Initialization workpanel and create a code page with the following code:

```
t=0;                        // initialize time
_view.resetTraces();        // clear old data
double u = _view.analyticFunction.evaluate(t);
_view.trace.addPoint(t,u); // add first xy-point
```

The initialization code clears old data from the trace and computes the first data point. This code is executed by the analytic function's action method and when the model is reset.

Exercise 3.4.
Run **TimeDependentFunctionPlotter** and observe the output. What happens to the output at $t = 10$? Why? □

3.5 FUNCTIONS OF TWO VARIABLES

A wave is a disturbance that propagates through space. Mechanical waves, such as water waves and sound, propagate through a material medium in contrast to electromagnetic waves, such as radio signals and light, which propagate through the vacuum. For a wave propagating in one dimension, we use a wave function $u(x, t)$ to represent the wave at position x and time t as shown in Figure 3.9. The disturbance can be mass density, pressure, or electric field depending on the physical context.

The simplest way to propagate a disturbance is to translate a function $f(x)$ without distortion. Translation by a distance d is implemented using

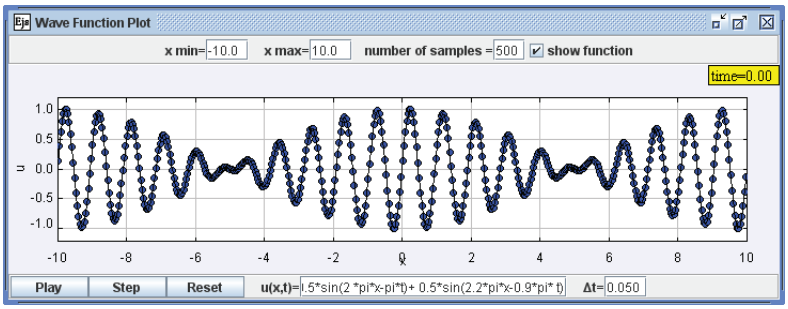


Figure 3.9: Two sinusoidal traveling waves with slightly different frequencies produce a wave function with a beat pattern which varies in space and time.

a coordinate translation

$$x' = x - d, \qquad \text{(coordinate translation)} \tag{3.5.1}$$

and the simplest wave function model propagates an initial disturbance $f(x)$ without distortion with a constant velocity v if $d = vt$. The time-dependent wave function is then

$$u(x,t) = f(x - vt). \qquad \text{(distortion-free propagation)} \tag{3.5.2}$$

Two prototypical functions are a Gaussian and a sinusoid. A Gaussian $f(x) = Ae^{x^2/2\sigma^2}$ is a pulse that has a smooth peak with height A and width σ. If we substitute $x - vt$ into the argument of the Gaussian function, we obtain

$$u(x,t) = Ae^{-(x-vt)^2/2\sigma^2}. \qquad \text{(Gaussian wave function)} \tag{3.5.3}$$

A sinusoid $f(x) = A\sin(kx)$ has multiple peaks where A is the height (amplitude) and k is the wavenumber. The wavenumber is a measure of the radians per unit length in the x direction. Because the sine function repeats when its argument increases by 2π, the wavenumber is related to its wavelength λ by $k = 2\pi/\lambda$. If we substitute $x - vt$ into the sine function, we obtain

$$u(x,t) = A\sin(k(x - vt)). \qquad \text{(sinusoidal wave function)} \tag{3.5.4}$$

The expression for a sinusoidal traveling wave (3.5.4) can be rewritten using different physical parameters. A common form of this expression is

$$u(x,t) = A\sin(kx - \omega t) \tag{3.5.5}$$

where $\omega = kv$ is the angular frequency and has units of radians per unit time. The angular frequency is related to the period $T = 2\pi/\omega$, which is the time it takes for the disturbance at a point to complete one oscillatory cycle.

The frequency $f = 1/T$ is the number of oscillations per second (Hertz) of the disturbance at a location x. Although a sinusoidal disturbance is a very common type of wave, there are many other wave functions, such as shock waves, that do not fit this functional form.

We now examine an existing wave function model that combines ideas from our previous function plotters. **WaveFunctionPlotter** allows us to input arbitrary wave functions $u(x,t)$ and displays the time evolution of the function. The expression for the wave function can be changed at run-time so that we can visualize a wide variety of disturbances. The simulation also displays markers (small circles) representing wave data taken at regular spatial intervals. These markers are created by setting properties in a trace object.

As explained previously, a Function element is an input field that can evaluate a mathematical expression (string) to produce a number. The name of the function is `waveFunction`, and we evaluate it using its `evaluate` method. Setting the Variable property to `x,t` in the inspector allows us to explicitly evaluate the expression using two arguments. The model's Fixed relations workpanel performs this evaluation using the following code:

```
double dx = (xmax-xmin)/n;                          // sample interval
double x = xmin+dx/2;                               // position
_view.trace.clear();                                // clear old data
for(int i = 0; i < n; i++){                         // loop to create new data
  double u = _view.waveFunction.evaluate(x,t);      // evaluate wave function
  _view.trace.addPoint(x,u);                        // add data to trace
  x += dx;                                          // increment position
}
```

The first two statements define and initialize local variables `dx` and `x` using global variables from the Variables workpanel. The third statement invokes the `clear` method to remove old data from the trace object. The remaining lines execute a loop that evaluates the function and adds points to the trace. Inspect the trace in the View workpanel and observe that its Connected property is `false` and its Marker Shape is `ELLIPSE`.

Inspect the *waveFunction* element in the control panel. The element's Variable property determines the order of the arguments when the function is evaluated in the loop. Inspect the analytic curve in the plotting panel. This element parses the `functionString` variable and draws a curve without having to set up a loop as was done in the Fixed Function Plotter.

The mathematical expression parser is very powerful. Unknown symbols within an expression are automatically associated with corresponding vari-

ables in the model. If the model defines variables named `amp`, `freq`, and `delta`, the user can entered the expression `amp*Math.sin(freq*x + delta*t)` into the function's text field and the function correctly evaluate the expression.

The Evolution workpanel for the model is simple. It consists of a single Java statement that increments time.

```
t += dt;
```

The time-dependent wave function is plotted because the code in the Fixed Relations workpanel is evaluated after every evolution step or whenever the user changes variables that are bound (connected) to properties in the graphical user interface.

Exercise 3.5. Wave function propagation
The default wave function in the model is a Gaussian described by $u(x, 0) = e^{-(x-3t)^2}$. In what direction does this wave function propagate? Change the function so that it propagates in the reverse direction. Change the function so that the Gaussian has twice the default width. □

Exercise 3.6. Sinusoidal wave function
Find a sinusoidal wave function with a speed of $v = 0.25$ to the right and with a wavelength $\lambda = 2.0$ and use the model to verify that your expression for the wave function is correct. □

Exercise 3.7. Frequency, wavelength, and phase
How does the disturbance change at a given spatial location? Run the model with $u(x, t) = \sin(0.5\pi x - \pi t)$ and set the number of sampling points to two in order to highlight the disturbance at two points. Verify that the oscillation frequency is $f = \omega/2\pi$ using the time display in the display's upper right hand corner. Note that the two points oscillate but that they are out of phase. □

The *superposition principle* states that the disturbance at a given location caused by two or more waves is the sum of the individual disturbances. We examine a common sinusoidal superposition in Exercise 3.8.

Exercise 3.8. Standing wave
When a wave encounters an obstacle, it is often reflected. Assume that a reflected sinusoidal wave has exactly the same shape as the incident wave. Enter $u(x, t) = 0.5\sin(2\pi x - \pi t) + 0.5\sin(2\pi x + \pi t)$ into the wave function model and describe the net disturbance. □

Exercise 3.8 illustrates a standing wave. There are points in space called *nodes* where the net disturbance is always zero. Midway between these nodes are points of maximum disturbance called *antinodes* where the oscillation amplitude is twice that of a single wave.

The wave function exercises and the end of chapter projects illustrate the rich variety of wave phenomena observed in Nature. Although distortion-free wave phenomena can be described analytically, the resulting wave functions are difficult to visualize without animation because they involve both space and time. Well-designed visualizations that show analytical solutions are useful because they reinforce the mathematics and provide feedback about the model's correctness and applicability.

> Because it is time consuming to type long expressions, *EJS* can save its parameters in a data file. Right-click in an empty section of a graph (don't right-click on an object such as a trace) and select the Save State input/output option to save your initial conditions. A saved data file can later be loaded into a simulation by right-clicking and selecting the Read State input/output option. A loaded file determines the initial state whenever the model is reset.

3.6 TOOLS

[Describe the data analysis, video capture, translation, and other tools built into Ejs. To be written.]

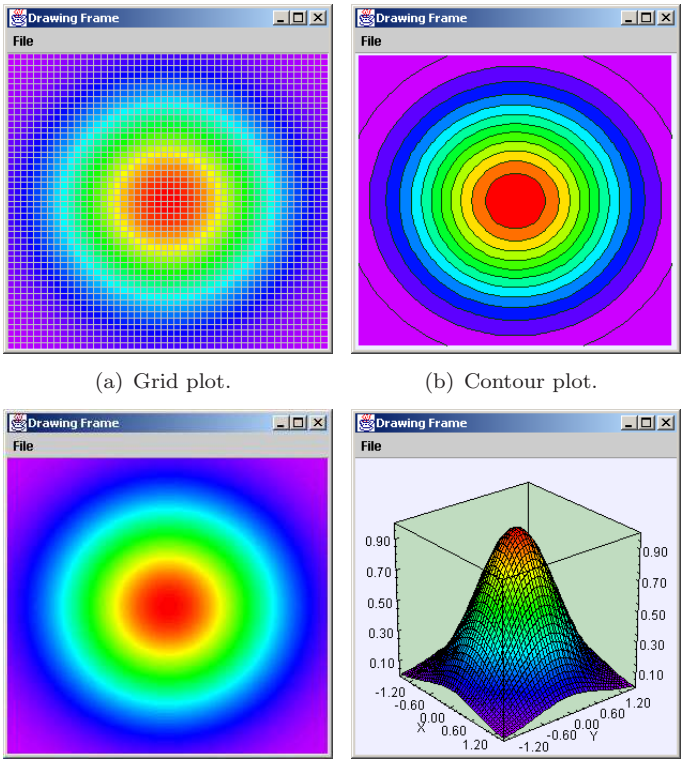
3.7 2D FIELDS AND ARRAYS

Imagine a sheet of metal that is heated at an interior point and cooled along its edges. In principle, the temperature of the sheet can be measured at any point. Calculating this temperature is a typical computational problem. A quantity, such as temperature, population density, or electrostatic potential energy, that is defined by a single number (scalar) at every point in space is a *scalar field*. Other types of fields are common. Consider the wind velocity outside your home. Because velocity is a vector, this situation defines a *vector field*. The following sections demonstrate how to display two-dimensional fields using elements on the *fields and plots* tab of the 2D Drawables palette.

Scalar and vector field visualizations almost always divide space into a discrete grid and arrays are a natural data type for storing this type of information. Although *EJS* creates many of the needed arrays behind the scene, we can do it ourselves if needed. For example, we can create a two-dimensional $n \times m$ array named u as follows:

```
int n=50;                          \\ number of rows
int m=40;                          \\ number of columns
double[][] u = new double[n][m]; \\ define array variable
```

An array is a block of computer memory that stores numbers, but keep in mind that computer memory is finite. A 1000×1000 array has 10^6 elements and consumes 8 megabytes of memory because each `double` array element requires 8 bytes. Three-dimensional scalar and vector field visualizations can require even larger blocks of memory because these visualizations use volume pixels (voxels) that are stored in arrays with three indices. Computing field values and drawing (rendering) pixels or voxels for large data sets is best done with specialized software and hardware. *EJS* is most effective when the grid is less than 100 points on a side.



(a) Grid plot. (b) Contour plot.

(c) Interpolated plot. (d) Surface plot.

Figure 3.10: Visualizations of the scalar field $U(x, y) = e^{-(x^2+y^2)}$.

The *Scalar Field 2D* and *Analytic Scalar Field 2D* elements on the *fields and plots* palette create two-dimensional visualizations using numeric data and analytic expressions, respectively, as shown in Figure 3.10. Both

elements use an array to store field data but the scalar field makes its array accessible whereas the analytic scalar field hides its array.

3.7.1 Analytic Scalar Fields

The electrostatic potential energy at a point is the energy required to move a test charge from that point to infinity. Our first scalar field model uses an analytic expression to display the electrostatic potential energy from an electrical dipole at the coordinate origin. This energy is a scalar that depends on the strength of the dipole p and on the distance r and angle θ from the center of the dipole. If the distance r is much larger than the charge distribution that is producing the dipole, the electrostatic potential energy can be approximated by

$$U(r,\theta) = \frac{kp\cos\theta}{r^2}\ , \tag{3.7.1}$$

where the electrostatic constant k depends on the choice of units. We choose units such that $kp = 1$ in our simple model.

A simple electric dipole is formed by two equal but opposite charges (+q and -q) separated by a distance d. Molecules, for example, often have a dipole moment even though they are electrically neutral because there is an uneven distribution of positive and negative charge associated to the atoms. The electric dipole moment p characterizes this type of charge distribution and is defined as the magnitude of the charge times the distance from the negative to the positive charge. The direction of the dipole is defined with respect to a line drawn from the negative to the positive charge. This direction is used when measuring the angle θ in the analytic expression.

Create a model named **DipolePotential** and save it in your working directory. Create a main frame with a plotting panel and select an analytic scalar field from the palette and add it to the plotting panel. Show the scalar field's inspector and set the Variable 1 property to `r`, the Variable 2 property to `theta`, and the Points 1 and Points 2 properties to `50`. Set the Coordinate System property to `POLAR`, the Z Data property to `cos(theta)/(r*r)`, the Autoscale Z property to `false`, and the Minimum and Maximum Z properties to -1 and 1, respectively. Set the Color Mode property to `REDBLUESHADE` and run the simulation.

The default field is shown using a grid of colored rectangles and the color of each rectangle represents the value of the field at the rectangle's center. Positive potential energies are shown in shades of red and negative potential energies are shown in shades of blue, but most if the grid

is black because the electric potential energy is close to zero. The culprit is, of course, the $1/r^2$ factor in (3.7.1). The dipole field at $r = 10$ is 100 times smaller than the field at $r = 1$ and it is difficult to display this large dynamic range on a computer monitor. The Expanded Z option allows us to enhance small magnitudes and suppresses large magnitudes to provide a better visualization of the field's structure. Set the Expanded Z property to `10`, set the Show Legend property to `true` to display the relation between color and field value, and run the simulation again. Note how the regions of positive and negative electric potential energy are distributed. Points along the vertical $\theta = \pi/2$ line have zero electrostatic potential energy because these points are equidistant from both the positive and the negative charge.

A grid plot is only one of many possible representations of a scalar field. Other representations are contour plots and surface plots, and these representations are explored in Exercise 3.9. When the simulation is running, you can also right-click within the panel and choose the *Elements options* menu item to show a table view of the data.

Exercise 3.9. Color palette
Customize the dipole potential model using the Type of Plot property to change the representation and the Color Mode property to change the color palette. Do this using the editor, the first icon to the right of the property values. Because the surface plot shows a scalar field using a color-coded 3D mesh with its own axes, you should set the plotting panel's Axis Type property to `null` when using this visualization. Which representation is best for large numbers of grid points? Small number of grid points? □

Exercise 3.10 demonstrates how a plotting panel sets its axis and gutters (margins) as its container is resized. Either the gutters or the axis maximum and minimum must be adjusted if we require that horizontal and vertical directions have the same number of pixels per unit.

Exercise 3.10. Gutters
What happens to the scale and the axis minimum and maximum values as the window is changed? Run the Dipole Potential and resize the window. The plotting panel automatically sets its x-axis and y-axis minimum and maximum values to the analytic field's default values. Exit the simulation and set the plotting panel's Square property to `true` and repeat. What happens to the axis minimum and maximum values now? Setting the Square property causes the plot to adjust the axis minimum and maximum values so that horizontal and vertical directions have the same number of pixels per unit and the scalar field no longer fills the entire plotting area. Exit the simulations and reset the Square property to `false` and set the Fixed Gutters property to `true` and repeat. What plotting panel property changes

when the frame size is adjusted? □

Return to the view workpanel and add a control panel to the main frame and add four radio button elements to this panel from the buttons and decoration palette. Name these elements `gridButton`, `interpolatedButton`, `contourButton`, and `surfaceButton`. Set the Text properties to the corresponding plot names, and set the Action On properties to `plotType = 0`, `plotType = 1`, `plotType = 2`, and `plotType = 3`, respectively. Navigate to the variables section of the Model workpanel and create an integer variable named `plotType`. Run the dipole potential simulation and observe that you can now change the representation of the field by selecting the appropriate radio button.

Exercise 3.11. Z scale
Add a number field to the control panel to set the Expand Z property. Add a check box to the control panel to show and hide the legend. □

3.7.2 Numeric Scalar Fields

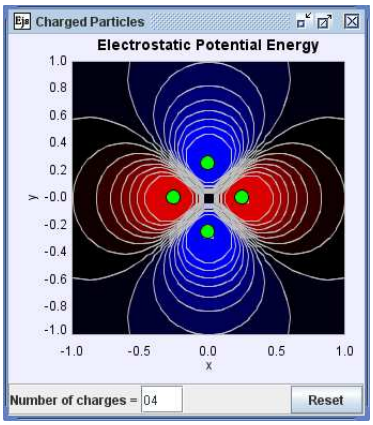


Figure 3.11: Four charged particles produce a quadrupole electrostatic field.

The analytic expression for an electrostatic dipole is simple because there are only two charges. What happens if we have multiple pairs of equal and opposite charges? Although it is possible to find an analytic expression for such a configuration using a multipole expansion, the resulting mathematical expression is cumbersome. A more direct way to obtain the electrostatic potential energy is to compute the scalar field by directly summing the contribution from each charge. The resulting scalar field is shown in Figure 3.11. This model requires that we know the location of the charges and the location of the grid points.

Create a model named **ChargedParticles** and save it in your working directory. Create a main frame with a plotting panel, select a *Scalar Field 2D* element from the palette, and add it to the plotting panel. Show the inspector and set the Z Data property to `potential`. The input field is pink because this variable does not yet exist. Rest the mouse on the property name to show its hint and note that the input variable must be a two-dimensional array of real numbers. Set the Autoscale Z property to `false`, the Expand Z property to 10, the Type of Plot property to `CONTOUR`, the Color Mode property to `REDBLUESHADE`, and the Minimum and Maximum Z proprieties to -10 and 10, respectively. Navigate to the variables section of the Model workpanel, create an integer variable `m` with a value of 64, and create the `potential[m][m]` array.

We next create variables to specify the location of the charged particles. Create `x`, `y`, and `q` variables with dimension $[n]$, where n is the number of charged particles and $q[i]$ is the charge of the i-th particle. Set the value of n to 4. Navigate to the Custom workpanel and define a method that positions the n charges on a circle with radius r.

```
// place n charges in a circle of radius r
public void initializeCharge() {
  double r= 0.25;             // distance from center
  double theta = 0;           // angle
  double dtheta= 2*Math.PI/n; // angle increment
  for(int i=0; i<n; i++){     // set position of n particles
    x[i]=r*Math.cos(theta);   // x position
    y[i]=r*Math.sin(theta);   // y position
    q[i]=(i%2==1)?-1:1;       // charges alternate in sign
    theta += dtheta;          // increment angle
  }
}
```

We use the modulus (remainder) operator `%` to alternate the sign of the charges. The `i%2` expression divides the counter by two and returns the remainder. The remainder will be zero or one if the loop counter `i` is an even or odd integer, respectively. The `?` operator is a shorthand form of an `if` statement. If the condition before the `?` operator is false, then the expression following the operator is evaluated. If the condition is true, the expression following the colon is evaluated.

Navigate to the Initialization workpanel and add the following statement to invoke the custom method.

```
initializeCharge();  // sets charge position
```

The electrostatic potential energy U at a distance r from a point charge q is $U = kq/r$, where k is a constant that depends on the units. We again choose units such that $k = 1$ in our model. The total electrostatic potential energy is the sum of n point charge electrostatic potential energies,

$$U = \sum_{a=1}^{n} \frac{kq_a}{r_a} \tag{3.7.2}$$

where r_a is the distance from the charge q_a to the grid point where the potential is being measured.

Navigate to the Fixed relations workpanel and add code to compute the electrostatic potential energy at the grid points.

```
for(int i=0; i<m; i++){                          // loop over grid x
  double xi= _view.scalarField.indexToX(i);      // x-coordinate of grid point
  for(int j=0; j<m; j++){                        // loop over grid y
    double yj= _view.scalarField.indexToY(j);    // y-coordinate of grid point
    potential[i][j]=0;                           // zero potential value
    for(int p=0; p<n; p++){                      // loop over charged particles
      double dx=xi-x[p];                         // grid point to charge x-separation
      double dy=yj-y[p];                         // grid point to charge y-separation
      double r=Math.sqrt(dx*dx+dy*dy);           // charge to grid point distance
      if(r!=0){                                  // check for singularity
        potential[i][j] += q[p]/r;               // add potential due to charge
      }
    }                                            // end of charge loop
  }                                              // end of y loop
}                                                // end of x loop
```

The fixed relation code has three nested loops. The `i` loop iterates through the grid's x-values and the `j` loop iterates through the grid's y-values. The inner loop iterates over the charged particles q_a with position (x_a, y_a), computes the distant to the grid point r, and sums the point-charge electrostatic potential energy q_a/r at the (x_i, y_j) grid point. Run the model to see that it displays the field from the point charges.

The charged particle model shows the electrostatic potential energy from `n` fixed particles but not the particles themselves. We now create an interactive model using draggable circles to show the particles and to enable mouse actions. Navigate to the *sets of 2D drawables* palette and create a *Particle Set* element named `particles` in the plotting panel. A set of elements is an array of objects whose properties can be set using arrays. Show the particles inspector and set the X property to `x`, the Y property to `y`, the Fill Color to `BLACK`, and the Dragable property to `true`. Sets of elements are easy to use because each element is bound (connected) to

a corresponding particle coordinate (x_a, y_a) and the default # Elements property is set by the dimension of the x variable. If a property value, such as the Fill Color, is constant, then the set assigns the same property value to all objects.

Exercise 3.12.
Run the Charged Particles model and drag the particles. What type of field is produced if two positive particles are placed at one location and the two negative particles are placed at another location? What type of field is produced if positive and negative particles overlap? □

How do we change the number of charged particles n when the model is running? The integer n is used to create (allocate) arrays of the proper dimension in the variables section of the Model workpanel and we must create arrays with the correct dimension if this number is changed. Return to the Custom workpanel and change the `initializeCharge` method to check if the length of the q array is equal to n.

```
public void initializeCharge() {
  // new code added here
  if(n!=q.length){            // has number of charges changed?
    x = new double[n];        // create arrays of proper length
    y = new double[n];
    q = new double[n];
  }
  // end of new code; remaining code is the same
  double r= 0.25;             // distance from center
  .
  .
```

You must also add an array check near the beginning of the fixed relation code.

```
if(n!=q.length){
  initializeCharge();
}
for(int i=0; i<m; i++){                      // loop over grid x
  double xi= _view.scalarField.indexToX(i);  // x value of grid point
// remaining code is the same
```

Exercise 3.13.
Modify the Charged Particles model by adding a number field that changes the number of particles and by adding a reset button that invokes the `_reset()` method. Figure 3.11 shows the completed user interface. What error message is produced if the array check is not included in the fixed relation code and where is this error message displayed? □

3.8 2D VECTOR FIELDS

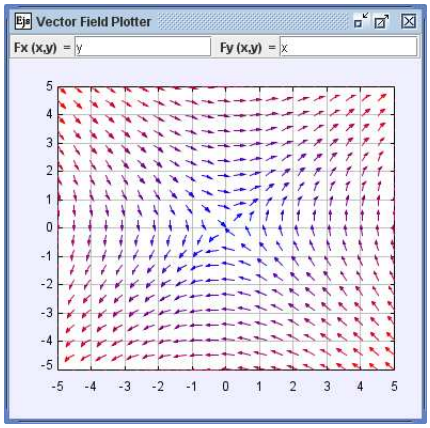


Figure 3.12: Fixed-length color-coded arrows are used to represent a vector field with a wide dynamic range. Pale blue and dark red represent small and large magnitude fields, respectively. These colors are converted to grayscale in black and white reproductions.

The *Vector Field 2D* and *Analytic Vector Field 2D* elements are similar to their scalar field counterparts except that two arrays are used to store the field data. We can either define two $n \times m$ array variables for the vector components or we can specify two analytic functions for the components. We construct an analytic model in this section and ask readers to build a numeric model in Exercise 3.14.

Create a model named **AnalyticVectorField** and save it in your working directory. Create a main frame with a plotting panel and select an analytic vector field from the palette and add it to the plotting panel. Show the field's inspector and set the Length property to 1 and the X and Y Component properties to `%xFunction%` and `%yFunction%`, respectively. Percent signs are required because we will use string variables to define analytic expressions. Navigate to the variables section of the Model workpanel and create `xFunction` and `yFunction` string variables with values `"y"` and `"x"`, respectively. Run the model and observe that all vectors have a length of one and that their colors change from blue to red as shown in Figure 3.12.

Using color rather than arrow length to represent a vector field's strength produces a more effective representation of magnitude over a wider range of values. Remove the Length property input in the inspector and run the model again. The vectors overlap because the default arrow length is the vector magnitude. The Color Component property in the appearance section of the inspector allows us to control the mapping of field strength to color. The default color component selects a color based on the field's mag-

nitude, but we can provide our own map. For example, setting the Color Component property to 1 uses the color that correspond to 1 for all arrows.

Return to the View workpanel and create a control panel in the top position. Add labels and text fields for the analytic functions and bind the text field Variable properties to the `xFunction` and `yFunction` variables. The complete user interface is shown in Figure 3.12.

Exercise 3.14. Charged particle electric fields
Modify the Charged Particles model so that it displays the electrostatic electric field from n particles. If we assume the particles are small, the total field at a grid point is the vector sum of point-charge electric fields $\mathbf{E}_a = (E_x, E_y)$. The vector components from a single point charge are

$$\begin{aligned} E_x &= \frac{kq_p\Delta x}{r_p^3} \\ E_y &= \frac{kq_p\Delta y}{r_p^3} \ , \end{aligned} \tag{3.8.1}$$

where r_p is the distance from charged particle q_p to the grid point and Δx and Δy are the x- and y-displacements from the particle to the grid point. Sum the vector components at each grid point to obtain the total electrostatic field. Are there regions close to an electric dipole configuration where the electric field is small? Modify the model so that it produces a ring of charged particles of the same polarity. Are there regions close to this configuration where the electric field is small? □

3.9 SUMMARY

At this point, we hope you have gained a feeling for how *EJS* and Java work together. We have introduced typical *EJS* elements and have studied their property inspectors. *EJS* inspectors are powerful tools for designing models because they provide two-way communication between the user interface and the model's global variables. Furthermore, inspector properties make reasonable assumptions about default values and most values need not to be set. It is easy to see what element properties are important for the implementation of a model because those property values have entries in their inspector. Although unused property values are usually uninteresting, you can always right click within an empty input field to examine the property's default value. For example, the default plotting panel left, right, top and bottom gutters for Cartesian Type 1 axes are $(45, 25, 25, 45)$, respectively, if the input field is empty. These values were selected because they provide the right amount of space for axis labels.

We have described many of the Java concepts that we will need to

implement our models. In particular, we have introduced arrays and objects and shown how they are used. We have also described many of the behind the scene operations that occur when we build a graphical user interface. Additional aspects of *EJS*, such as 3D modeling and the ordinary differential equation editor, will be taught by example but we will no longer describe the details of how to construct a user interface.

A final word about simplicity of design. *EJS* makes it easy to create user interface elements for every parameter and every initial condition in a model. Resist the temptation to include every possible parameter and to provide input fields for every variable. Good models focus the user's attention on parameters or combinations of parameters that significantly affect the outcome. The best models are those that capture the relevant science with a small number of parameters. This Occam's Razor description of the *best model* is an uncomputable task. In the final analysis, the best model is the one that works best for you and the people you expect to use it.

PROBLEMS

Problem 3.1 (Special Functions)**.** The *OSP* library distributed within *EJS* contains a numerics package that defines special functions of interest to scientists and mathematicians. One such special function is the Hermite polynomial class. Create a model that plots the n-th Hermite polynomial on the domain $[-2, 2]$ by importing the `org.opensourcephysics.numerics.Hermite` class into your model. (Use the import field in the simulation information panel.) You can evaluate the n-th polynomial at x using the following statement:

```
double y = Hermite.evaluate(n,x); // evaluates n-th Hermite polynomial at x
```

(Advanced) Create a model that displays quantum mechanical simple harmonics oscillator energy eigenfunctions.

Problem 3.2 (Parametric Curves)**.** Parametric curves are often used to represent curves that are not functions. Modify the Analytic Function Plotter to display a parametric curve defined in terms of functions $x = f(s)$ and $y = g(s)$ of an independent parameter s that is not a Cartesian coordinate. Test your model using $x = \cos s$ and $y = \sin 5s$.

Problem 3.3 (Nyquist's Theorem)**.** How often should a recording, such as a music CD, sample sound to accurately reproduce the sound? Run the Wave Function model with $u(x, t) = \sin(2\pi x - \pi t)$ without displaying the wave

function. Sample the wave function at 21 points on the interval $-10 < x < 10$. What wavelength do you observe in the sampled data? Which way does the wave appear to move? How many samples are required to reproduce the shape of the wave function?

This problem illustrates an important information theory concept known as Nyquist's theorem. The theorem states that the measurement interval must be smaller than half the oscillation interval to reproduce the shape of a wave function. For temporal measurements the interval must be smaller than half the period T. Because the human ear can perceive sounds from 20 hertz to 20 kilohertz, music studios sample sound at a minimum of 40 kilohertz.

Problem 3.4 (Diffraction)**.** Diffraction occurs when a wave encounters an obstacle or passes through an aperture. The resulting diffraction pattern is most pronounced when the wavelength is on the order of the size of the diffracting object. For example, monochromatic light generated by a laser passing through a rectangular aperture with width a and height b produces a pattern with maxima and minima when a screen is placed behind the aperture. The intensity pattern $I(x, y)$ on a screen that is placed far from the aperture is

$$I(x, y) = I_0 \left(\frac{\sin ax}{ax} \right)^2 \left(\frac{\sin by}{by} \right)^2 , \tag{3.9.1}$$

where I_0 is the light intensity incident on the aperture and (x, y) is a screen coordinate in appropriate units. Model this intensity pattern using a scalar field plot.

Problem 3.5 (Wave Superposition)**.** Create a model that illustrates the wave superposition principle. Allow the user to input two mathematical expressions using two input fields and display these wave functions and their sum. Use this model to show the interference (beat) pattern between two waves with slightly different frequencies as shown in Figure 3.9. For example, enter $u_1(x, t) = \sin(2\pi x - \pi t)$ and $u_2(x, t) = \sin(2.2\pi x - 0.9\pi t)$ into the model and describe the net disturbance. Note that the wave function shape has both a fast oscillation and a slow oscillation. Why? Observe the minima in the wave function. In what direction does the fast oscillation minima move? The slow oscillation minima?

-

PART 2

From Free Fall to Chaos