

## Chapter Five

### Continuous Time Models

The purpose of computing is insight, not numbers. *Richard Hamming*

We study models for continuous systems which can be expressed using ordinary differential equations. We cover interesting models in dimensions one and two, but also higher-dimensional problems that involve differentiating arrays. We introduce powerful graphic and numeric tools embedded in *EJS*, and study new visualizations such as direction fields and phase-space plots.

#### 5.1 THE COOLING COFFEE PROBLEM

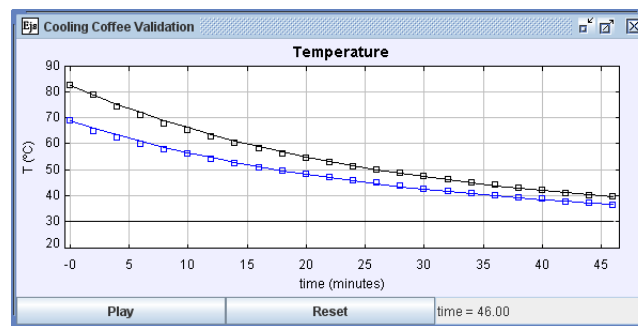


Figure 5.1: Comparison of Newton's law of proportional cooling with experimental data of the cooling of a cup of black coffee (top plot) and of coffee with cream. The model data (continuous line) fits well with the experimental data (square marks). Project 5.1 shows how to read an experimental data file.

You are about to give a short presentation of your latest computer simulation to your professor when two freshly brewed cups of coffee arrive. You both decide to wait until the end of the presentation (5 minutes) to drink your coffees but your professor immediately adds some cream to her

coffee while you wait until the end of your presentation. Your presentation went well but you wonder who will drink a hotter coffee. Curiosity is a basic ingredient of science and asking what, why, and how things happen leads to a deeper understanding of how nature works. Your question makes a perfect case for modeling, the central theme of this book.

In 1958 two Cornell University engineering students presented a report title “The Mechanisms of Cooling Hot Quiescent Liquids” in which they studied the effect of adding cream to a cup of hot coffee. Because typical brewing temperature for coffee is 85C (185F) and drinking temperature is 62C (143F), they studied the effect of adding cream when the coffee is served or adding it just before it is consumed. This is not a trivial problem because a hot body exchanges heat with its surroundings through the simultaneous processes of conduction, convection, evaporation, and radiation. Newton argued that since the thermal energy is proportional to the volume and the energy loss is proportional to the exposed area, the time of cooling is proportional to the diameter. Larger objects therefore cool more slowly. Laplace, Helmholtz, and Kelvin extended Newton’s model by considering gravitational energy and radiation to estimate the age of the Sun to be well over 20 million years. Although the discovery of nuclear fusion greatly increased this age estimate, the balance between cooling and thermonuclear fusion is of fundamental importance in stellar models. Today we use advanced heating, cooling, and energy transport models to debate the impact of global warming. Considering in detail the processes in these climate change models requires advanced supercomputer-based algorithms and is beyond the scope of this text. We wish to lay the groundwork for such studies and we start by studying the cooling-coffee problem. We create a simple model of how objects cool and implement this model using different numerical algorithms. We then use this simulation with different conditions to predict the final temperature.

If the temperature difference between an object and its surroundings is not too large, Newton’s law of proportional cooling states that the rate of change of the liquid temperature is proportional to that difference. This relationship can be expressed in mathematical form using the ordinary differential equation

$$\dot{T}(t) = -k(T(t) - T_r). \quad (5.1.1)$$

where  $T(t)$  is the temperature of the liquid at time  $t$  and the dot on top of it to the left of the equal sign indicates its derivative with respect to time, i.e.  $\dot{T}(t) = \frac{dT(t)}{dt}$ . The reference temperature of the surroundings  $T_r$  is assumed to remain constant in our simple model. Finally,  $k$  is a constant that depends on several factors, such as the geometry of the problem (a bigger surface of contact between liquid and air favors faster cooling), the mass of the liquid, and the particular liquid considered (its specific heat).

Newton's law of proportional cooling is not a fundamental law, but an empirical relationship that works well in practice, and is an example of a continuous time model. Continuous time models are those in which time is considered to flow uniformly causing the state variables of the system (in our problem, the temperature of the liquid) to change. In this chapter we describe continuous time models which can be modeled using ordinary differential equations (ODEs). Our expectation is to be able to ascertain the evolution in time of the temperature of the liquid from equation (5.1.1) and the knowledge of its initial temperature (the initial conditions).

### Exercise 5.1.

If you have studied calculus, show that

$$T(t) = T_r + (T_0 - T_r)e^{-k(t-t_0)} \quad (5.1.2)$$

is the solution of (5.1.1) and therefore provides the desired evolution of the temperature in time where  $T_0$  is the temperature at time  $t = t_0$ . You can substitute (5.1.2) into (5.1.1) but you can also derive the solution by separating the variables in the differential equation and integrating.  $\square$

Simulating a continuous time model for which we know an explicit analytic solution, such as our cooling problem, is relatively easy. We select a sequence of successive time instants  $t_0 < t_1 < t_2 < \dots$ , and compute and display the state at these instants using the explicit solution. In other words, we reformulate the continuous model as a discrete model but the time intervals can be made arbitrarily small. Discrete models for which the time interval cannot be made arbitrarily small are discussed in Chapter 6.

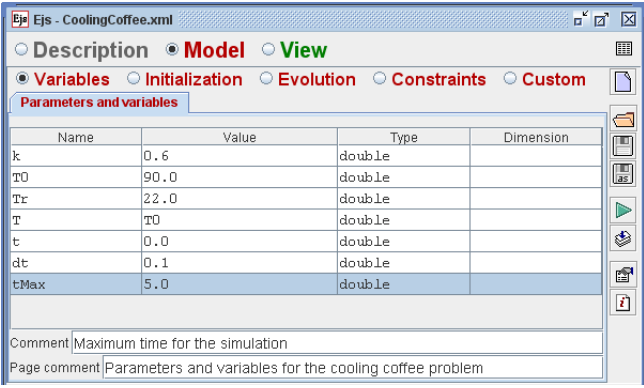
The Cooling Coffee model declares the page of variables shown in Figure 5.2. We set  $k = 0.6$ ,  $T_0 = 90$ , and  $T_r = 22$  and we simulate the model from  $t = 0$  to  $t = 5$ . Temperature is measured in Celsius degrees and time is measured in minutes. (What are the units for  $k$ ?) We have chosen a fixed discretization step  $dt$  of 0.1 minutes to compute the temperature of the coffee at equally spaced instants of time  $t_k = t_0 + k dt$ .

Enter the following code in the Evolution workpanel.

```
t += dt; // Increment the time
if (t>=tMax) _pause(); // stop if maximum time is exceeded
```

The Constraints workpanel contains a code page with the closed-form solution (5.1.2) of the differential equation.

```
T = Tr + (T0-Tr)*Math.exp(-k*t); // closed-form solution
```



The screenshot shows the Ejs interface for 'CoolingCoffee.xml'. The 'Model' tab is selected, and the 'Parameters and variables' section is active. A table lists the following parameters and variables:

Name	Value	Type	Dimension
k	0.6	double	
T0	90.0	double	
Tr	22.0	double	
T	T0	double	
t	0.0	double	
dt	0.1	double	
tMax	5.0	double	

Below the table, there are two text fields: 'Comment' with the value 'Maximum time for the simulation' and 'Page comment' with the value 'Parameters and variables for the cooling coffee problem'.

Figure 5.2: Table of variables for the cooling coffee problem.

The view is simple and consists of a plotting panel with traces for the reference temperature and for the temperature of the liquid. Running the simulation produces an exponentially decreasing curve as seen in Figure 5.3.

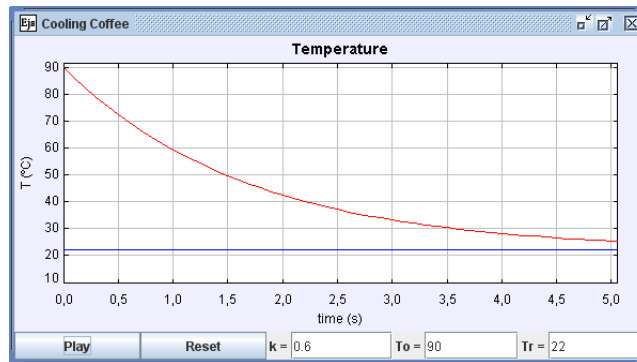


Figure 5.3: Typical exponentially decreasing temperature for Newton's law of proportional cooling.

The fields to the right bottom of the user interface display the parameters of the model and their actions initialize the system so that you can compare different responses.

To compare different responses, you can capture a snapshot of the view for each set of parameters. Right-click on an empty area of the plotting panel to bring in its popup menu. Select from this menu the option "Capture the screen > Snapshot" and save the resulting graphic file. The plotting panel's Print Target property has been configured so that this menu option captures the contents of the `mainFrame` element.

### Exercise 5.2.

Add a button to the view with an action that adds cream to the coffee. If

the amount of time it takes to add the cream is small, the temperature of the coffee-cream mixture  $T'$  can be computed by assuming that heat lost by the coffee  $\Delta Q_- = Mc(T' - T)$  is equal to the heat gained by the cream  $\Delta Q_+ = mc(T' - T_c)$ . Assume that cream is served at  $T_c = 5\text{C}$  (41F) and that the heat capacity  $c$  of coffee is approximately that of cream. Setting  $Q_+ = -Q_-$  and solving for the final temperature  $T'$  gives a simple equation

$$T' = \frac{MT + mT_c}{M + m} \quad (5.1.3)$$

where  $M \approx 250$  grams is the mass of coffee and  $m \approx 25$  grams is the mass of cream.  $\square$

Not all ODE problems have such a simple solution. The well-developed theory of ordinary differential equations states that, under reasonable conditions on the differentiability of the expressions involved, ODEs have a unique solution for realizable initial conditions. But asserting the existence of a solution and actually finding it are different things! Except in a few happy cases (see for instance [?]), ODEs are difficult or even impossible to solve analytically. For example, the ODE

$$\dot{x} = 3x \sin x + t, \quad (5.1.4)$$

has no known explicit solution.<sup>1</sup> In cases such as this, we must resort to numerical or graphical techniques to approximate the solutions of the equations to the required level of accuracy.

When computing a particular solution of an ordinary differential equation we must also providing an initial condition for the state at time  $t_0$ ,

$$\mathbf{x}(t_0) = \mathbf{x}_0. \quad (5.1.5)$$

An initial condition together with a corresponding differential equation is known as an *initial value problem*. Many numerical algorithms for solving initial value problems exist and which one to use for a given problem depends on both the cost of the computation and on the nature of the problem itself. In the following sections we describe simple methods for continuous time models which can be described by systems of ordinary differential equations of the form

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t). \quad (5.1.6)$$

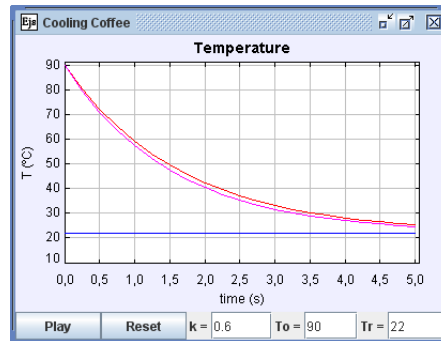
We use vector notation (indicated by the use of bold font) for the state of the model  $\mathbf{x} = \{x_1, x_2 \cdots x_n\}$  in order to cover, later in the chapter, higher-dimensional problems. The dimension of the problem is that of the state vector. The vector function  $\mathbf{f} = \{f_1, f_2 \cdots f_n\}$  expresses the rate of change of the state as  $n$  functions of time and the state itself. A special, and very

---

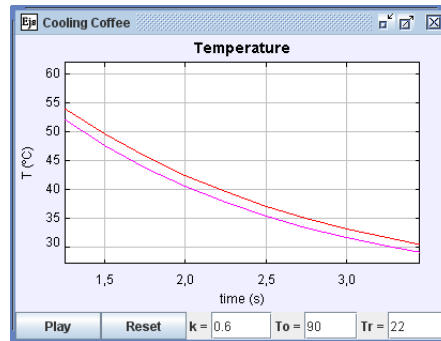
<sup>1</sup>We will frequently follow the custom of omitting the explicit dependence on  $t$  of the state variable  $x$ .

important type of differential equation is that of *autonomous systems*, in which  $\mathbf{f}$  has no explicit dependence on time. Autonomous systems, also called *dynamical systems*, appear frequently in the description of continuous physical processes, such as the cooling coffee problem described above. We discuss the properties of these special systems in Section 5.9.

## 5.2 EULER METHOD



(a) Euler method and exact solutions.



(b) Zoom of a portion of the graph.

Figure 5.4: Comparison of numerical and exact solutions of the cooling coffee problem for  $k = 0.6$ ,  $T_0 = 90$ , and  $T_r = 22$  in the first five minutes. The solution computed using Euler method (the lower trace) approximates but does not match exactly the analytic solution. The right image shows a zoomed portion of the graphs.

The standard approach to finding the numeric solution of an initial value problem is that of *difference methods*. In them, we discretize the independent variable (usually the time) and try to compute, if only approximately, the value of the state at a time  $t_0 + \Delta t$ , for some  $\Delta t > 0$  where  $\Delta t$  is called the *step size*. The simplest such method uses the first two terms of

the Taylor expansion of the solution:

$$\mathbf{x}(t_0 + \Delta t) = \mathbf{x}(t_0) + \Delta t \dot{\mathbf{x}}(t_0) + \frac{(\Delta t)^2}{2!} \ddot{\mathbf{x}}(t_0) + \dots \quad (5.2.1)$$

and the fact that  $\dot{\mathbf{x}}(t_0) = \mathbf{f}(\mathbf{x}(t_0), t_0)$ , to obtain a first approximation,  $\mathbf{x}_1$ , of  $\mathbf{x}(t_0 + \Delta t)$ :

$$\mathbf{x}_1 = \mathbf{x}_0 + \Delta t \mathbf{f}(\mathbf{x}_0, t_0). \quad (5.2.2)$$

To simulate the evolution of our model, we iterate algorithm (5.2.2) to advance repeatedly the state of the model in time. This algorithm, known as **Euler's method**, was first considered by Leonhard Euler in the XVIII century.<sup>2</sup>

The **CoolingCoffeeEuler** model extends our previous **CoolingCoffee** model by using Euler's method to compute a numerical approximation of the solution. The model defines a new trace in order to display a new variable called **Teuler**, which is initialized to  $T_0$ . The code in the Evolution workpanel is now:

```
Teuler = Teuler - dt*k*(Teuler-Tr); // Euler approximation
t += dt; // increments time
if (t>=tMax) _pause(); // stops if maximum time is exceeded
```

When the evolution executes this code, the model steps from the current state **Teuler** at time **t**, to the new state at **t+dt** given by the new value of **Teuler**. The analytic solution is computed using constraints exactly as before.

If we plot the solution of the system in the range  $[0, 5]$  computed with a value of 0.25 for the step size, **dt**, together with the analytic solution, we see (left plot of Figure 5.4) how the solution provided by Euler's method closely approximates the analytical solution. However, if we zoom in the picture (right image of Figure 5.4) we observe a small discrepancy of about 1.8 degrees in the central part of the plot.

To zoom the picture, right-click within the plotting panel, select the plotting panel the *Elements options* from the popup menu, and navigate to the submenu "Plotting Panel > Zoom > Zoom in". Then, click and drag with the left button of the mouse to select the rectangle of the plane to zoom. You can also compare the numerical data of both traces. To obtain the data of a trace in tabular form, select the element options for a trace and navigate to the "Show data in table" submenu. Compare the entries for both traces around  $t = 2.5$ .

---

<sup>2</sup>Euler was born in Basel, Switzerland in 1707 and died in St. Petersburg, Russia in 1783.

Euler's method does not provide the exact value of the solution at  $t_0 + \Delta t$ , but an approximation to it. It can be proved that, if  $\Delta t$  is small enough, the approximation can be made as accurate as desired. There is, however, a computational limit to the accuracy because of the finite precision of computer arithmetic. In addition, a smaller value of  $\Delta t$  requires more steps to advance the evolution of the system through a unit time interval. Hence, accuracy comes at the price of additional computation.

### Exercise 5.3.

Run the **CoolingCoffeeEuler** model with  $dt=0.25$  and with  $dt=0.025$  and record the maximum difference between the analytical and numerical results. By how much does the error decrease? Note that the simulation also runs more slowly. This additional time is due both to the increased number of computations and data points stored in the trace which is displayed in the plotting panel. You can remove this second problem by using the inspector to set the Skip property of the traces to 10 so that only one out of every ten points is stored.  $\square$

A straightforward theoretical analysis shows that the total (global) error produced by Euler's method decreases linearly with  $\Delta t$  when advancing the model from  $t_{initial}$  to  $t_{final}$ . Euler's method is therefore termed a method of order one. A numerical method that reduces the total error quadratically with  $\Delta t$  is said to be second order. In general, the order of a method is the power-law relationship between the time step and total error. For example, *EJS* has an eighth-order method that reduces the error by a factor of  $256 = 2^8$  when the time step is reduced by a factor of two. Although Euler's method is not used in actual computations (because it produces a poor approximation, unless  $\Delta t$  is prohibitively small), it is taught all around the world for its simplicity and because it paves the way for further, more sophisticated algorithms.

## 5.3 ONE STEP METHODS AND THE ODE EDITOR

There are several ways of improving the accuracy of Euler's method. In Exercise 5.4 we outline one possible method that uses more terms of the Taylor expansion of the solution.

### Exercise 5.4.

Modify the evolution code of **CoolingCoffeeEuler** to add the third term on the right of equation (5.2.1), i.e.  $\frac{(\Delta t)^2}{2!} \ddot{T}(t_0)$ . The second derivative of  $T$  can be computed by differentiating the original relationship  $\dot{T} = -k(T - T_r)$ . Check that running the improved model with a value of  $dt = 0.25$  produces results comparable to that of the Euler method with  $dt = 0.025$  with no appreciable loss of speed. This method is called the *three-terms Taylor*



*method* and is a second-order method because the error on a unit interval decreases quadratically with  $\Delta t$ .  $\square$

Although one can create more accurate numerical methods by taking subsequent terms in the Taylor expansion, the derivatives involved become more and more complicated and error-prone. Moreover, the differentiation and coding effort done for one problem can not be reused for a different ODE. Over the years, mathematicians looked for numerical methods of order higher than one that only require the evaluation of the rate function  $\mathbf{f}$ , and are therefore easy to reuse. Among the most popular and easy to implement for simulation purposes are the so-called *one step methods*. These methods evaluate the rate function at multiple points in the interval  $[t_0, t_0 + \Delta t]$  and then combine the results to provide a good approximation of the solution at  $t_0 + \Delta t$ . Although a better approximation requires more evaluations of the rate function for a single step, the extra computations pay off because the higher accuracy implies that the step size  $\Delta t$  can be increased, sometimes dramatically, reducing the total number of computations in a given interval of time  $[t_{initial}, t_{final}]$ . We describe some of the most popular one step methods in the appendix at the end of this chapter on numerical methods.

Writing a simple, fixed step size implementation of some of these methods is not too difficult. However, numerical methods reach their maximum efficiency only when implemented using advanced numerical techniques that involve automatically estimating the error, adapting (changing) the step size, and interpolating the resulting points to produce the final state. In order to make it easy to use such advanced solution algorithms, *EJS* has a built-in ODE editor that allows you to enter the differential equations in a natural form and select the numerical method desired. *Easy Java Simulations* automatically creates Java code that uses the *Open Source Physics* library [?] to solve the equations. We now illustrate how to use the editor to implement a variation of the cooling coffee problem with changing outside temperature.

The **CoolingCoffeeEditor** simulation uses the ODE editor in the Evolution workpanel to implement the model of a liquid that cools in a changing outside temperature  $T_r(t)$ ,

$$\dot{T}(t) = -k(T(t) - T_r(t)). \quad (5.3.1)$$

Although the resulting differential equation is linear, finding the solution requires computing an integral which may not be analytically solvable, depending on the complexity of the function  $T_r(t)$ . We must therefore resort to a numeric method.

Figure 5.5 shows how equation (5.3.1) is entered into the ODE editor.

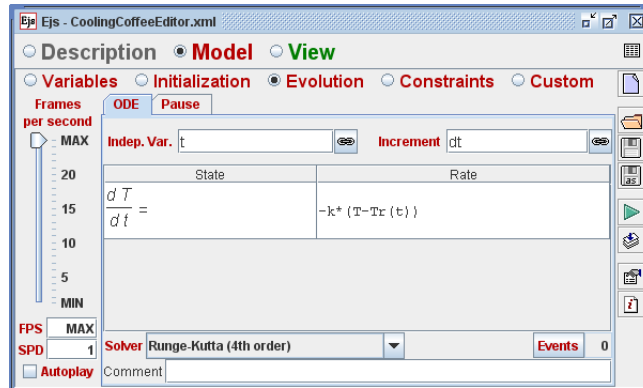


Figure 5.5: The ODE editor of *EJS* used to specify the equation of the cooling coffee problem with variable reference temperature. The rate of the state variable  $T$  on the left column is computed using the formula on the right column.  $\text{Tr}()$  is a method declared in the `Custom` panel of the model.

The Independent Variable property at the top of the panel is used to specify the independent variable of the differential equation. This variable can be any continuous variable but is often the time as in the current model. The Increment property specifies the step size of our discretization. The central part of the editor contains a table showing the formula that computes the rate of each of the state variables. To enter a differential equation, we double-click on the left column and type the corresponding state variable ( $T$  in our problem). The editor then shows the rate in the familiar  $\frac{dT}{dt}$  form. Alternatively, right-clicking on the state cell allows us to select the state from the list of model variables of `double` type. The rate is specified by double-clicking in the right column and typing the Java expression required for the computation. In our case, the code is `-k*(T-Tr(t))`, where `Tr` is now a method (function) defined in the `Custom` workpanel of the model with the code:

```
public double Tr (double time) {
    return 22.0 + 10.0*Math.sin(time*Math.PI); // periodic fluctuations
}
```

Notice that, although time is a global variable `t`, the  $t$  variable in the ODE editor is a local variable with possibly different values. Numerical methods compute rates at different (intermediate) instants of time in order to achieve better precision. For example, the fourth order Runge–Kutta method evaluates the rate four times when advancing the system from  $t_0$  to  $t_0 + \Delta t$ . Because the state variables in an ODE editor are not the global variables with the same name, it is crucial to follow the rule of passing the

ODE editor's state variables as parameters to methods in the rate column of the editor.

Finally, the lower part of the ODE editor allows us to select one of the one-step methods available in *EJS* to solve the problem. The classical fourth-order Runge–Kutta method chosen for this example provides an almost perfect match with the correct solution even for large values of the step size.

If the method selected uses an adaptive algorithm, a field will appear allowing us to enter the tolerance (or maximum local error) desired for the computation. The method will then use smaller internal step sizes, if needed, to make sure that each step has an error smaller than the tolerance. A detailed explanation of how adaptive methods work is provided in the appendix at the end of this chapter. The **Events** button is used for problems with discontinuities in the equation and is discussed in Chapter 7.

Solving a given ODE with different methods and values of the step size is always recommended. The number of different algorithms used today accounts for the fact that no method is superior in all circumstances. Selecting an appropriate step size can help us obtain an accurate solution without wasting computer power and one of the big advantages of the ODE editor in *EJS* is that it makes this experimentation very easy.

#### Exercise 5.5.

Run the **CoolingCoffeeEditor** model for a constant outside temperature (i.e.  $T_r(t) = 22.0$  for all  $t$ ) with different step sizes and compare the solution obtained with the analytic one. See how large can  $\Delta t$  be for a Runge–Kutta method. Check the order of the Runge–Kutta method to be four by computing how the error decreases with  $\Delta t$ .  $\square$

#### Exercise 5.6.

Model the temperature of an object that is periodically heated and cooled in a discontinuous manner. Assume that the heating-cooling cycles has a frequency  $f$ .

```
public double Tr (double time) {
    if(Math.sin(2*Math.PI*frequency*time)<0){
        return 22.0 + 10.0;
    }else{
        return 22.0 - 10.0;
    }
}
```

A discontinuous  $T_r(t)$  function can confuse a numerical ODE algorithm. Are some algorithms more successful than others in handling discontinuities?  $\square$

## 5.4 ODE FLOW AND DIRECTION FIELDS

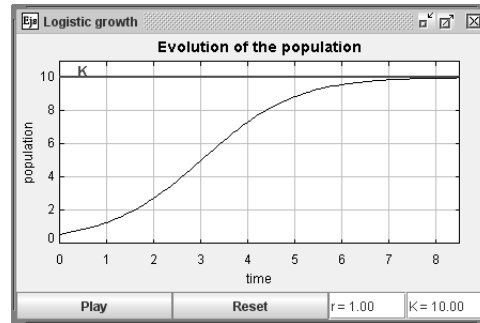


Figure 5.6: Evolution of a continuous time model for population growth. The variable which represents the population (in normalized units) changes continuously in time with a rate which depends on its current value and on the value of the parameters  $r = 1$  and  $K = 10$ . The system reaches 99% of its carrying capacity after approximately 8 time units.

Differential equations are frequently used in biology to model population dynamics. A very simple model of the evolution of the population  $N$  states that the species will reproduce according to the Malthusian law

$$\dot{N} = rN, \quad (5.4.1)$$

where the growth rate  $r$  is a given positive parameter. Equation (5.5.1) can be easily solved to show an exponential increase of the population which would soon drive the species into overpopulation problems unless there is infinite food and space. More sophisticated models predict exponential growth for small populations but include an overcrowding term that limits the growth of the population due to decrease and the scarcity of resources. The *logistic model* for a population  $N$  (expressed in suitable units) predicts this behavior and be written as

$$\dot{N} = rN - \frac{r}{K}N^2 \quad \text{or} \quad \dot{N} = rN\left(1 - \frac{N}{K}\right) \quad (5.4.2)$$

where population  $N(t)$  is a continuous function of time,  $r > 0$  is the growth rate for small populations, and  $K > 0$  is the ecosystem's carrying capacity. Figure 5.6 shows that the carrying capacity is reached, starting from an initial value of 0.5, after approximately eight time units if  $r = 1.00$ .

Although the solution for an initial value problem, such as the logistic equation, with a particular set of initial conditions is useful, we often wish to understand the qualitative behavior of all the possible solutions of a differential equation. Even if we have an analytic solution, it may be complicated and it may be unclear how the solution depends on the initial conditions. A better approach is to study the rate of change (flow) as a function of the

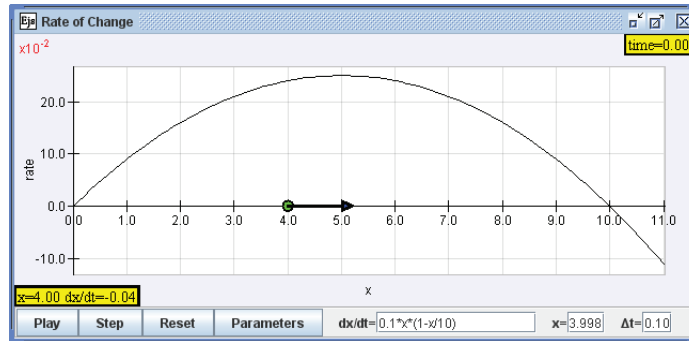


Figure 5.7: The one-dimensional phase space flow of the logistic equation.

state variables. The **Ode1DFlow** model implements this visualization for a single ordinary differential equation of the form

$$\dot{x} = f(x). \quad (5.4.3)$$

The ODE 1D Flow rate graph (Figure 5.7) shows the logistic function  $f(x) = 0.1x(1 - x/10)$  using the generic  $x$  to represent the population. (What is the value of  $K$ ?) The graph contains a draggable red circle that represents the current value of  $x$  and has an arrow that shows the direction of  $\dot{x}$ . If the rate is positive, the arrow is to right; if negative, the arrow is to the left. This arrow introduces us to phase space flow, a concept that allows us to study the geometry of a differential equation. The time development graph (see Figure 5.6) displays particular solutions  $x(t)$  when the model is run.

### Exercise 5.7.

Load the one-dimensional ODE flow model into *EJS* and run the default (logistic) simulation. Stop the time evolution, drag the red circle past the carrying capacity  $K = 10$ , and restart the time evolution and compare the two solutions. Can the circle ever move past the carrying capacity? What information that is available in the solution plot is missing from the rate plot? □

Points where  $f(x) = 0$  are called *fixed points* or *equilibrium points* because the rate of change is zero and the value of  $x$  remains constant. Figure 5.7 shows that the logistic equation has two fixed points. The fixed point at  $x = 0$  is an *unstable* fixed point because solutions in its vicinity always move (flow) away from it. The fixed point at the carrying capacity  $x = K$  is a *stable* fixed point because solutions in its vicinity move (flow) toward it.

**Exercise 5.8.**

Use the one-dimensional ODE flow model to identify and classify the fixed points for  $\dot{x} = \sin x$  on the interval  $[-4\pi, 4\pi]$ .  $\square$

**Exercise 5.9.**

Use the one-dimensional ODE flow model to identify and classify the fixed points for  $\dot{x} = x - x^3$  on the interval  $[-1.5, 1.5]$ .  $\square$

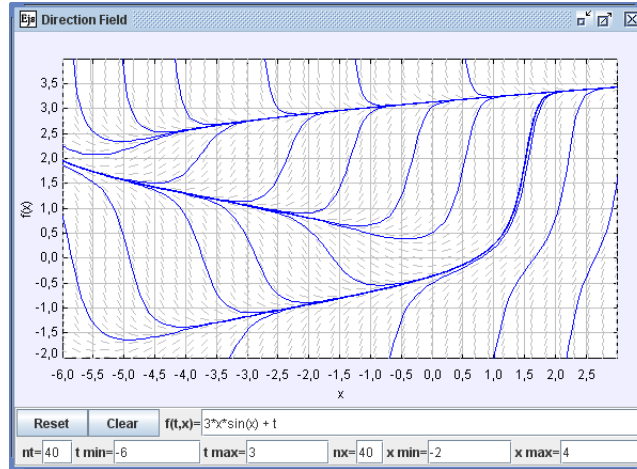


Figure 5.8: A grid of small line segments showing the direction field of the ODE  $\dot{x} = 3x \sin x + t$  in the range  $[-6, 3] \times [-2, 4]$ . Multiple solution curves have been computed and superimposed on the field.

A graphical technique which is commonly used for one-dimensional equations with rate function  $f(x, t)$  that also depends on time (non-autonomous ODE) is that of plotting the *direction field* of the ODE. This visualization consists of drawing a grid of line segments in the  $(t, x)$  solution plane. Each segment has a constant length and a slope given by  $f(x, t)$ . Since the solution curve obeys  $\dot{x}(t) = f(x(t), t)$ , the curve must be tangent to these line segments at each of the field's grid points. When the grid is refined, the qualitative behavior of the solution curve can frequently (but not always) be sketched. Figure 5.8 shows the direction field and a collection of solution curves for the ODE  $\dot{x} = 3x \sin x + t$  in the  $[-6, 3] \times [-2, 4]$  solution plane.

The **DirectionFieldPlotter** model shows the direction field representation of a differential equation by superimposing ODE trajectories on an analytic vector field with components  $(1, f(x(t), t))$ . The line segments show the ODE rate of change and the **nt** and **nx** indexes indicate the number of grid points in the  $t$  and  $x$  directions. The visualization uses a fixed color map with constant length line segments as described in Section 3.8. Trace elements are used to plot the solution curves.

Contrary to the usual way of starting the evolution using a button action, this model's evolution is started using a mouse action. Every point within the plotting panel is a possible initial conditional for the differential equation and the model generates the solution curve that passes through a point when the user clicks on that point within the panel. In order to show the complete solution, uses separate differential equations to advancing forward and backward in time from the starting time  $t_0$ . Forward and backward solutions are recorded in the `solutionForward` and `solutionBackward` Trace elements.

$$\begin{aligned} \dot{x}_f(t) &= f(x, t_0 + t) && (\textit{forward}) \\ \dot{x}_b(t) &= -f(x, t_0 - t) && (\textit{backward}) \end{aligned} \quad (5.4.4)$$

Note that the independent parameter  $t$  is set to zero at the start of the evolution and that this value must be added and subtracted from  $t_0$  to generate the forward and backward solutions, respectively.

The plotting panel's On Release action executes code that controls the animation by invoking the following custom method:

```
public void compute_solution() {
    // return if not left-clicking so as not to interfere with pop-up menu
    if (_view.plottingPanel.getMouseButton() != EjsConstants.LEFT_MOUSE_BUTTON) return;
    _pause(); // stop the animation
    t = 0; // reset the animation time
    t0 = _view.plottingPanel.getMouseX(); // get initial t
    xf = xb = _view.plottingPanel.getMouseY(); // get initial x
    _view.solutionForward.moveToPoint(t0+t,xf); // add first point to forward trace
    _view.solutionBackward.moveToPoint(t0-t,xb); // add first point to backward trace
    _play(); // start the animation
}
```

We use the `getMouseButton()` method to identify the mouse button that was released so as not to interfere with the plotting panel's pop-up menu. If the left button is pressed, the code stops the animation and reads the mouse coordinates to initialize the model's state variables. The code then adds initial values to the Trace elements that record the forward and backward solutions and starts the animation. The Evolution workpanel advances the both  $x_f$  and  $x_b$ . The evolution is stopped if the solution can no longer be seen within the plotting panel using a constraint that performs the following bounds check.

```
if(((t0+t>tmax)|| (xf>xmax)|| (xf<xmin)) && // forward outside bounds
    ((t0-t<tmin)|| (xb>xmax)|| (xb<xmin))) // backward outside bounds
    _pause(); // stop if both solutions are out of bounds
```

The Direction Field model is a good example of how to start and stop the evolution from within a model and we encourage you to study it carefully.

**Exercise 5.10.**

Predict (sketch the direction field for logistic model. Pay particular attention to the direction field at the fixed points. Test your prediction using the **DirectionFieldPlotter** model. The logistic model is autonomous. How does this affect the direction field?  $\square$

**Exercise 5.11.**

Use the **DirectionField** simulation to draw the direction fields for the following ODEs in the given regions of the plane. Sketch the solution curves before clicking within the simulation to compute the solution.

- $\dot{x} = -k(x - T_r)$ , for  $k = 0.03$  and  $T_r = 22$  in  $[0, 5] \times [10, 90]$ . (The cooling coffee problem.)
- $\dot{x} = (1 - t)x - t$ , in  $[-2, 4] \times [-4, 2]$ .
- $\dot{x} = (x - 3)(x + 1)/(1 + x^2)$ , in  $[-5, 5] \times [-3, 5]$ .
- $\dot{x} = -x \cos t + \sin t$ , in  $[-10, 10] \times [-10, 10]$ .
- $\dot{x} = 3t^2/(3x^2 - 4)$ , in  $[-2, 2] \times [-2, 2]$ .

$\square$

The last equation in Exercise 5.11 shows that numeric methods are not infallible. The solver fails to find the right solutions at the turning points (as indicated by vertical field segments). The reason is that the ODE is incorrectly defined for points in the vertical lines  $x = \pm\sqrt{4/3}$ .

**Exercise 5.12.**

Change the solver of the ODE editor to the adaptive method **Runge-Kutta-Fehlberg 5(4)** and test it with the last of the equations in Exercise 5.11. See that the *EJS* console prints a non-convergence warning when the trajectories reach the line  $x = \pm\sqrt{4/3}$ . Fixed step methods simply ignore the problem and try to step forward at any cost (and provide wrong solutions).  $\square$

## 5.5 PREDATOR AND PREY

Higher dimensional differential equations appear naturally in continuous models of real-life processes. One of the nicest examples of a two-dimensional



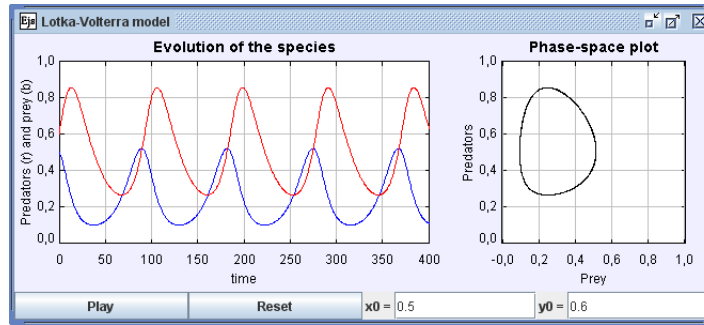


Figure 5.9: Typical oscillating behavior of the Lotka–Volterra model for the predator (top plot) and prey system. The phase-space (right) plot clearly shows that the oscillation repeats itself in time periodically.

model comes from the field of population dynamics in Biology: the Lotka–Volterra model for predators and prey.

Suppose a population of a given species that lives in a region with plenty of food. We again start with the Malthusian law for the evolution of the number of individuals

$$\dot{x} = a x, \quad (5.5.1)$$

where the growth rate  $a$  is a given positive parameter. Equation (5.5.1) predicts an exponential increase of the population which would soon drive the species into overpopulation problems unless there is infinite food and space.

Notice that we are considering the population  $x$  to be a continuously varying quantity, even when the number of individuals is certainly an integer. This assumption is licit when the number of individuals is sufficiently high and we re-scale the problem in a suitable way. For instance,  $x = 1$  might actually mean that there are one million individuals. A change of one individual can then be considered approximately a continuous change in the model.

But populations rarely live in isolation. In particular, this species shares its living space with a second species which predares on it. Without predation, the number of individuals of the predator species  $y$  would follow the rather discouraging dynamics given by equation

$$\dot{y} = -c y, \quad (5.5.2)$$

for some constant  $c > 0$ . This dynamics will clearly soon drive the species to extinction.

Predation changes the dynamics of both populations and can help reach a sustainable ecological state. If we assume that both populations

are constantly mixed in space, then predation takes place continuously and turns the dynamics into a coupled two-dimensional system given by

$$\begin{aligned}\dot{x} &= ax - bxy, \\ \dot{y} &= -cy + dxy.\end{aligned}\tag{5.5.3}$$

The positive constants  $b$  and  $d$  account for the predation rate of the encounters between individuals of both species, and the increase in the reproduction rate of the predators produced by the nutritive value of the predation, respectively. This system of ODEs is called the Lotka–Volterra model for predators and prey.

Different examples of predator-prey systems have been considered in real applications. Historically, the first Lotka–Volterra model was applied by the Italian mathematician Vito Volterra<sup>3</sup> to explain why the percentage of the catch of two different categories of fishes in the Mediterranean Sea changed during World War I. The predator group of species were selachians (sharks, skates, rays, ...) which prey on other, more desirable food fish harvested by ships from the port of Fiume, Italy, during the years 1914–1923. A very nice historical and theoretical note of this study can be found in [?] and [?].

Creating a simulation that solves Lotka–Volterra’s equations is straightforward using the ODE editor in *EJS*. The **LotkaVolterra** simulation implements equations (5.5.3) in the editor (see Figure 5.10) and solves them to display the evolution of the populations in time.

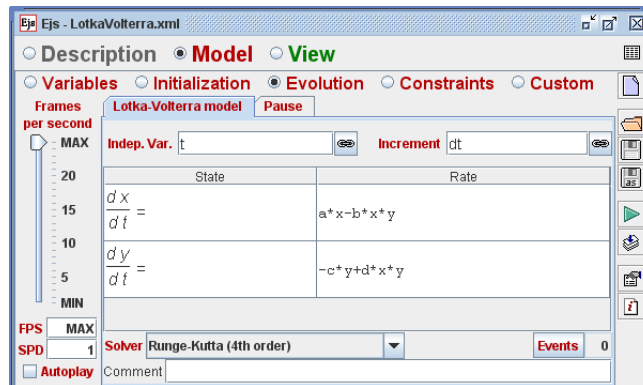


Figure 5.10: Lotka–Volterra equations in the ODE editor of *EJS*.

The view of the simulation shown in Figure 5.9 follows a pattern already familiar to us and we do not discuss its implementation in detail.

<sup>3</sup>Vito Volterra was born in Ancona, Italy in 1860 and died in Rome in 1940. Alfred Lotka, who devised the predator and prey model independently, was born in Lemberg, Austria–Hungary (now L'viv, Ukraine) in 1880 from US parents, and died in New York, in 1949.

Notice however that we are not displaying the solution curves of the system. The solution would be a three-dimensional curve given by points of the form  $(x, y, t)$ . Instead, the left panel shows the *component graphs* of the solution, given by points of the form  $(x(t), t)$  and  $(y(t), t)$ , and the right panel shows the *trajectory* of the solution in phase-space, given by points of the form  $(x, y)$ . This display is similar to what we did in the mass and spring example of Chapter 2.

The component graphs provide information of how the population of each species changes in time and is very close to how fishermen would annotate the populations of fish in a time register. We can see the alternate extrema of both species. The ups and downs of the predators follow closely those of the prey, as one would expect.

The phase-space (also called state-space) plot is a mathematical sophistication consisting in projecting the three-dimensional solution curve into the plane span by the state coordinates. Although this projection loses the time information (we cannot tell *when* the trajectory passes through a given  $(x, y)$  point), it provides important information which is not easily appreciated in the component graphs or the three-dimensional curve. For instance, the right view of Figure 5.9 immediately suggests that trajectories are periodic, i.e. they repeat themselves periodically in time. Volterra proved mathematically that all phase-space trajectories of equations (5.5.3) are periodic. The period of the trajectories, i.e. the minimum time for which they repeat themselves, varies slightly depending on the trajectory. The sole exceptions are the trajectories that start at points  $(0, 0)$  and  $(c/d, a/b)$ , which are constant. Constant trajectories are known as *equilibrium solutions* or *stationary states*.

Additional information can be obtained from phase-space plots for autonomous systems. (The Lotka–Volterra model is an autonomous system because the rates of  $x$  and  $y$  do not depend on time.) Trajectories in phase-space of autonomous systems can't cross each other, which limits the kind of possible behavior for trajectories. Also, if a trajectory of an autonomous system passes twice through a given point, it becomes periodic. Autonomous systems are discussed in more detail in Section ??.

**Exercise 5.13. Volterra's law of averages.**

Volterra also proved analytically that, despite having different periods, the average predator population over a period is precisely  $c/d$  and the average of the prey population  $a/b$ . The averages are given by the integrals

$$\bar{x} = \frac{1}{T} \int_0^T x(t) dt, \quad \bar{y} = \frac{1}{T} \int_0^T y(t) dt, \quad (5.5.4)$$

where  $T$  is the period of the particular trajectory considered. Show this fact by computing numerically this average. [Hint: Modify simulation **Lotka-Volterra** so that, given initial conditions  $(x_0, y_0)$ , it counts the accumulated population (times the step size) at each integration step until the trajectory passes again the line  $y = y_0$  from below. Divide then the counter by the time elapsed and compare to the value predicted by Volterra.]  $\square$

#### Exercise 5.14. Sensitivity to initial conditions in modified Lotka–Volterra systems

Suppose the reproduction rate of the prey undergoes periodic changes (according to seasons, say). Change the system so that  $a = a_1 + a_2 * \cos(ft)$ , where  $a_1$  denotes a fixed rate and  $a_2$  and  $f$  define a periodic fluctuation. Show that the system now can be very sensitive to initial conditions. Use for instance  $a_1 = 0.172$ ,  $a_2 = 0.25$ , and  $f = 0.3$ . Plot the trajectory with initial conditions  $x_0 = 0.46501$ , and  $y_0 = 0.40811$  until  $t = 800$  and then compare it with the trajectory which starts at the same  $x_0$  but with  $y_0 = 0.40810$  (a difference of only  $10^{-5}$ !). Sensitivity to initial conditions, together with erratic behavior are footprints of chaos.  $\square$

## 5.6 NEWTONIAN MECHANICS

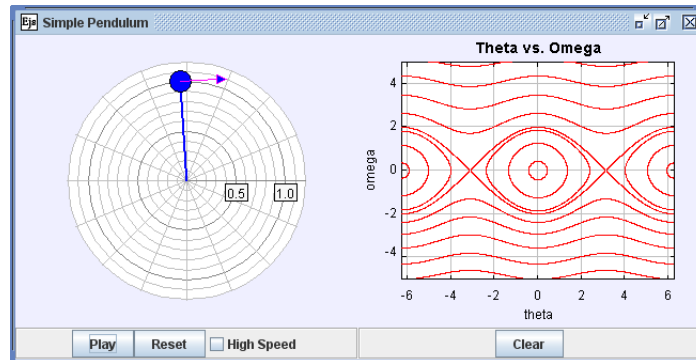


Figure 5.11: Motion of an undamped, undriven simple pendulum. The phase-space portrait on the right shows a collection of different trajectories. The current trajectory is the lowest one in the phase-plane, in which the pendulum whirls in one direction for ever.

A common source for higher-dimensional ordinary differential equations is the field of Newtonian mechanics. Newton's Second Law states that the acceleration of a point particle of mass  $m$  is related to the sum of all forces acting on it,  $\mathbf{F}$ , according to the relation

$$\mathbf{F} = m \mathbf{a}. \quad (5.6.1)$$

If the total force  $\mathbf{F}$  can be computed from the position  $\mathbf{x}$  and velocity  $\dot{\mathbf{x}}$  of the

particle and the time  $t$ , equation (5.6.1) expresses a differential relationship of the form

$$\ddot{\mathbf{x}} = \frac{1}{m} \mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t). \quad (5.6.2)$$

This equation is a second order differential vector equation with the dimension of the position variable  $\mathbf{x}$ . Second and higher order problems can be converted into ordinary (first order) differential equations of the form (5.1.6) introducing additional variables for the derivatives of the state vector. Thus, equation (5.6.2) for a particle with coordinates  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  can be rewritten in the desired form by introducing a new state vector

$$\mathbf{x}' = (x_1, v_1, x_2, v_2, \dots, x_n, v_n) \quad (5.6.3)$$

where  $v_i = \dot{x}_i$ . Notice that this procedure doubles the dimension of the state vector so we have twice as many differential equations to solve. But it makes the problem accessible using the numerical methods for the solution of first-order equations presented in this chapter.

We saw a first example of a mechanical system describing the motion of a mass and spring in Chapter 2. That motion displayed the characteristics of a linear oscillator. We consider now the simplest example of a non-linear oscillator. A simple pendulum is a physical abstraction consisting of a point mass  $m$  which oscillates at the end of a rigid massless rod of fixed length  $L$ .

Massless rods and point masses are abstractions that do not exist. However, a dense small sphere at the end of a light thin rod is a good approximate for the idealized pendulum.

When the pendulum is at rest, it will hang vertically from the pivot point. When displaced from the vertical and released from rest (with no initial velocity), the pendulum will oscillate in the plane which contains the vertical line through the pivot point and the initial position. This restriction converts a problem in space into a problem in the plane. Because of the rigidity of the rod, the motion of the mass is restricted to a circumference. Polar coordinates  $(r, \theta)$  are therefore more appropriate for this problem than the usual cartesian coordinates  $(x, y)$ . The radial component  $r$  is fixed and equal to the length of the rod  $L$ . The interesting dynamics is related to the change of the angular coordinate  $\theta$ .

Newton's Law for planar rotation states that the angular acceleration  $\ddot{\theta}$  of an object is proportional to the torque  $\tau$  applied to that object,

$$\tau = I \ddot{\theta}. \quad (5.6.4)$$

The constant of proportionality  $I$  is known as the moment of inertia and

can be shown to be  $I = mL^2$  for a point mass that is at a distance  $L$  from the point of rotation. Applying Newton's Second Law for rotation to the pendulum leads to the following second-order differential equation

$$\ddot{\theta} = -\frac{g}{\ell} \sin(\theta). \quad (5.6.5)$$

Following the procedure indicated above, we introduce the new variable given by angular velocity  $\omega = \dot{\theta}$  to turn this second-order equation into an equivalent first order system

$$\begin{aligned} \dot{\theta} &= \omega \\ \dot{\omega} &= -\frac{g}{\ell} \sin(\theta). \end{aligned} \quad (5.6.6)$$

This is the first order system of ODEs for the (undamped, undriven) simple pendulum. Additional terms must be introduced when the system includes frictional or driving forces.

The **SimplePendulum** model implements equations (5.6.6) using the ODE editor of *EJS* and shows the evolution of the pendulum using both a realistic representation of the motion of the pendulum and a phase-space diagram. The model declares three separate pages of variables for organizational reasons. See Figure 5.12.

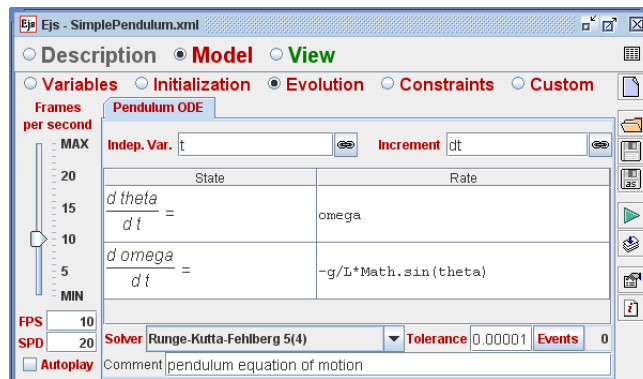


Figure 5.12: Three different tables help organize the variables of the simple pendulum model. Pages of variables are processed from left to right.

All variables in these tables have global visibility irrespective of the page in which they have been declared. A single exception is that variables can be used in the *Value* column of other variables only if they have been declared earlier in the same table or in a previous page of variables. Pages of variables are processed from left to right.

The ODE editor is used to state and solve the differential equations (5.6.6).

Figure 5.12 shows we have chosen an adaptive algorithm with a tolerance of  $10^{-5}$  to compute the solution.

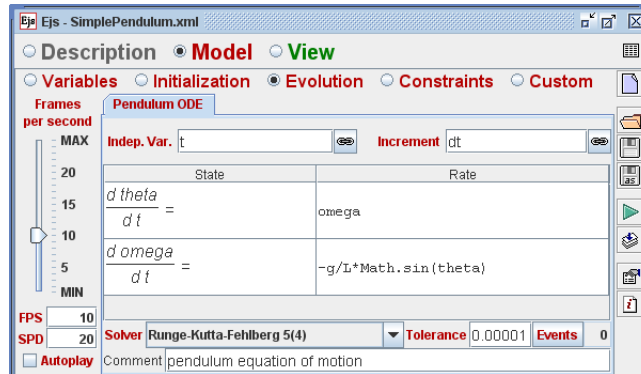


Figure 5.13: First order system of differential equations equivalent to the second order problem  $\ddot{\theta} = -\frac{g}{L} \sin(\theta)$ .

Because the equations of motion are solved in polar coordinates, the model includes a page of constraints with the following code to convert from polar to cartesian coordinates.

```
x= L*Math.sin(theta);
y = -L*Math.cos(theta);
vx = omega*L*Math.cos(theta);
vy = omega*L*Math.sin(theta);
```

These cartesian coordinates are used to position the bob and its velocity vector in the left plotting panel of Figure 5.11. This plotting panel displays (for aesthetic reasons, mainly) polar axes according to the value of its *Axis Type* property. The property inspector displayed in Figure 5.14 shows the location of the bob is given using the constrained *x* and *y* coordinates.

The polar-cartesian dichotomy forces us to provide an action for the view element which displays the bob. When the user drags the bob to a new position, the action code computes the correct value of the `theta` angle.

```
theta = Math.atan2(x,-y);
// length is constant
omega = 0.0;
t = 0.0;
```

The `Math.atan2(a,b)` method computes the angle subtended by the given coordinates in the  $[0, 2\pi]$  interval which is used as the new value of `theta`.

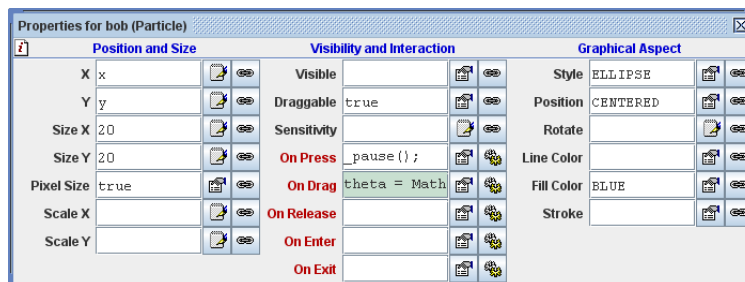


Figure 5.14: Property inspector for the bob view element. Because its position is given by cartesian coordinates, action code is required to update the model polar coordinates.

The action code also resets the time and sets the angular velocity to zero so that the motion starts from rest. Because  $L$  is not changed, constraints (which will be called automatically by *EJS* after the action is executed) will correct the values of the  $x$  and  $y$  coordinates so that the bob remains at the correct distance from the pivot.

The right plotting panel of the view of this simulation is prepared to display a phase-portrait of the simple pendulum. A phase-portrait is a visualization of a moderately large collection of different trajectories in phase-space, and provides a good deal of information of the qualitative behavior of the solutions of the system. The user can click on any point in phase-space and obtain the trajectory through that point.

We have chosen to display the trajectories in the phase-portrait as a collection of separate points, rather than using a connected trace. The reason is that displaying a complete phase-portrait such as the one shown in Figure 5.11 requires plotting a large number of points. Although a single trace can also display several trajectories, too many points can make the computer slower and even run out of memory. The *DataRaster* element we used instead accepts individual points within prescribed minimum and maximum coordinates and prepares an off-screen bitmap image which it dumps to the screen when the view is refreshed. The memory usage is minimal and the drawing very fast. Points are added to the data raster by linking its  $X$  and  $Y$  properties with the values of `theta` and `omega`, as shown in Figure 5.15.

We make the phase-space appear periodic in the angular dimension by identifying points in the vertical sides of the rectangle through the `periodicAngle` custom method.

```
public double periodicAngle(double angle) {
```



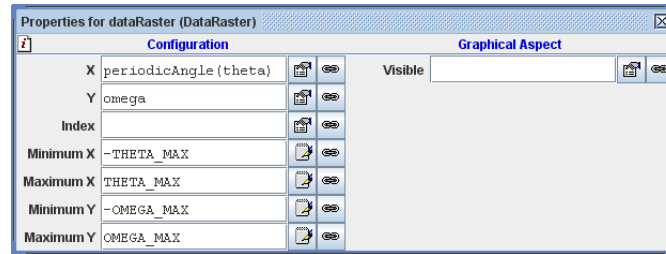


Figure 5.15: Property inspector for the data raster view element. The angular (horizontal) coordinate is made periodic to identify the vertical sides of the rectangle.

```

while (angle>THETA_MAX) angle -= 2*THETA_MAX;
while (angle<-THETA_MAX) angle += 2*THETA_MAX;
return angle;
}

```

The only disadvantage of this approach is that we will need to compute many trajectory points, close to each other, to get the impression of a continuous curve. For this reason, we have chosen a small increment of time, 0.01. But, because refreshing the screen (and, in general, graphic activities) takes much more computer time than number-crutching, we have set the parameter of *Steps Per Display* (SPD) in the evolution to 20, meaning that the view will be refreshed only after 20 evolution steps. The *highSpeed* check box element in the simulation interface can even increase the SPD parameter to 200, causing trajectories to be computed much more efficiently (although the realistic pendulum motion becomes wild).

### Exercise 5.15.

Locate the equilibrium positions of the pendulum in the phase-plane. They correspond to the pendulum up and down position with zero angular velocity. The lower position is stable (though not asymptotically stable) and the upper one unstable. Although difficult to locate exactly, the phase-portrait suggests that there are two trajectories in which the bob tends to the upper position. However, it would take an infinite time for the bob to reach this limit position. These trajectories are called *separatrices* because they separate regions of the phase-plane where the trajectories show different qualitative behavior.  $\square$

## 5.7 A CHAIN OF OSCILLATORS

We introduced simple harmonic motion caused by a linear oscillator in the **MassAndSpring** model of Chapter 2. We now consider a linear array of coupled springs forming a chain of oscillators as displayed in Figure 5.16.

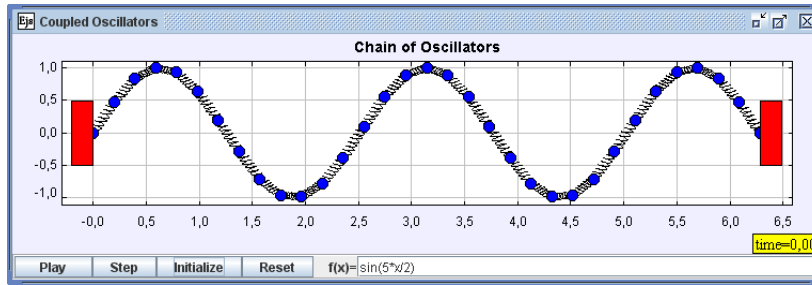


Figure 5.16: A chain of coupled linear oscillators modeled with *EJS*. The configuration displayed here corresponds to a normal mode where every particle moves sinusoidally with the same frequency.

This model can be used to study the propagation of waves in a continuous medium and the vibrational modes of a crystalline lattice.

The **OscillatorChain** model contains 31 coupled oscillators equally spaced within the interval  $[0, 2\pi]$  and with fixed ends. Let  $y_i = y(x_i, t)$  represent the displacement of a particle with horizontal position  $x_i$  along the oscillator chain. Because it is assumed that the particles do not move in the  $x$ -direction, we need only consider forces in the vertical direction. The force  $F_i$  on the  $i$ -th particle depends on the relative vertical displacement between that particle and its nearest neighbors and can be written as

$$F_i = -k[(y_{i+1} - y_i) - (y_i - y_{i-1})]. \quad (5.7.1)$$

where the Hooke's Law  $k$  constant is the same for all springs.

The main page of variables of this model, displayed in Figure 5.17, declares one-dimensional arrays for the dynamic variables  $x$ ,  $y$  and  $v$ .

Name	Value	Type	Dimension
n	33	int	
x	0.0	double	[n]
y	0.0	double	[n]
v	0.0	double	[n]
t	0.0	double	
dt	0.5	double	

Comment: Vertical coordinates of the particles  
Page comment: Dynamic variables of the oscillator chain

Figure 5.17: Dynamic variables for the oscillator chain. Arrays of  $n$  length are used for the coordinates and velocities of the particles.

Declaring an array is certainly appropriate for this problem due to the

large number of particles involved. The initialization of the horizontal and vertical positions of the masses,  $x$  and  $y$ , requires the following code in an initialization page of the model.

```

_pause();
t = 0;
double x0 = 0;
for(int i=0; i<n; i++) {
    x[i] = x0;
    y[i] = _view.function.evaluate(x0);
    v[i] = 0;
    x0 += dx;
}

```

This code distributes the masses according to an initial pulse given by the expression introduced by the user in the **Function** element of the view.

The most interesting new feature of this model is the use of the ODE editor of *EJS* with one-dimensional arrays. Figure 5.18 shows that the ODE editor accepts differential equations where the state variables are given by elements of one-dimensional arrays. In this case, we just need to type the state variables  $y$  and  $v_y$ , and the editor will add to them the suffix  $[i]$ . This suffix defines a dummy index  $i$  which can be used in the **Rate** column to provide the rate of the given component of the state array.

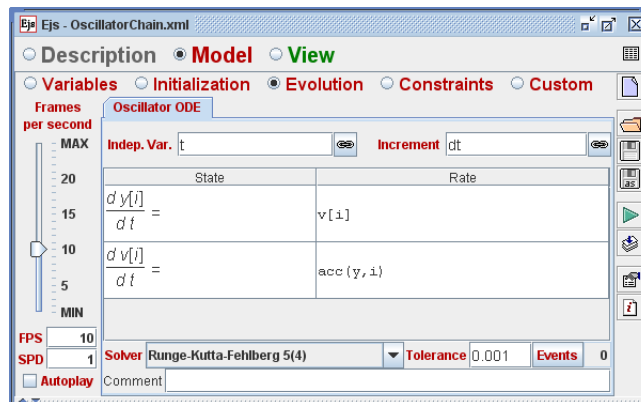


Figure 5.18: The ODE editor of *EJS* used to declare a system of ODEs with one-dimensional arrays as state variables.

In our case, the rate of the  $i$ -th state component  $y[i]$  is simply the  $i$ -th component of the velocity array,  $v_y[i]$ . The rate of the  $i$ -th component of the velocity array is provided by a custom method defined as

```
public double acc(double[] y, int index) {
```

```

    if (index==0 || index==n-1) return 0;
    else return k*(y[index-1]+y[index+1]-2*y[index]);
}

```

which corresponds to equation (5.7.1) (for unit masses). The fixed end points of the chain have zero acceleration. Finally, a page of constraints computes the lengths of the springs which will be used for the visualization.

```

for(int i=0; i<n-1; i++) springDy[i] = y[i+1]-y[i];

```

The view of the simulation uses sets of drawable elements to display the  $n$  particles and  $n-1$  springs which form the chain. Of particular interest is the `On Drag` action property of the `particles` element. The code

```

double xp=0;
for (int i=0; i<n; i++) {
    x[i] = xp;
    xp += dx;
}
_resetSolvers();

```

reinitializes the `x` array to equidistant values to counteract the possibility that the user drags the masses horizontally. The `_resetSolvers()` predefined method of *EJS* is required only in case we plan to use a numeric solver for the ODE which uses interpolation. Using an interpolator solver when there is a high number of equations involved may improve performance. Interpolation solvers keep an internal copy of the ODE state that they use intelligently to optimize the performance of the solver. The `_resetSolvers()` method must be called whenever the user changes the state of the simulation so that interpolators can update their internal state to match that of the system. Solvers which use interpolation are clearly marked in the `Solver` combo box.

One way of understanding a lattice of  $N$  coupled oscillators of length  $L$  and mass  $M$  is to study the motion of its normal modes. A normal mode is a special configuration (state) where every particle moves sinusoidally with the same frequency. The  $m$ -th mode  $\Phi_m$  of the oscillator chain of length  $L$  is

$$\Phi_m(x) = \sin \frac{m \pi x}{L}. \quad (5.7.2)$$

The system stays in a single mode and every particle oscillates with constant angular frequency  $\omega_m$  if the oscillator chain is initialized in a single mode.

$$\omega_m = \frac{4k}{M} \sin \frac{m \pi}{2N}. \quad (5.7.3)$$

Normal modes are important because an arbitrary initial configuration can be expressed as sum of normal modes.

**Exercise 5.16.**

Run the **OscillatorChain** model and experiment with different normal modes. The  $m$ -th normal mode of this system can be observed by entering  $f(x) = \sin(mx/2)$  as the initial displacement. Do all particles oscillate with the same frequency in given normal mode? Do all normal modes have the same oscillatory frequency? Are there a finite or an infinite number of normal modes? (That is, what happens to the oscillator chain as  $m$  increases.)

□

Wave propagation can be studied by entering a localized pulse or by setting the initial displacement to zero and dragging oscillators to form a wave packet. An interesting and important feature of the **OscillatorChain** model is that the speed of a sinusoidal wave along the oscillator array depends on its wavelength. This causes a wave packet to disperse (change shape) and imposes a maximum frequency of oscillation (cutoff frequency) as is observed in actual crystals.

**Exercise 5.17. Dispersion exercise here.**

□

## 5.8 A MINI SOLAR SYSTEM\*

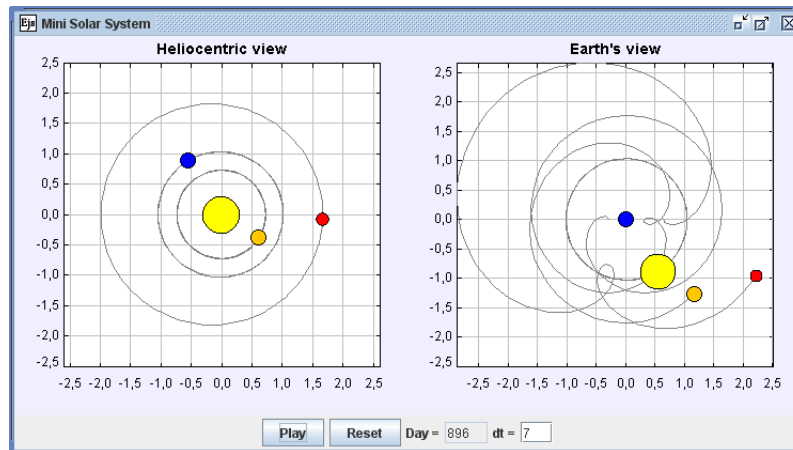


Figure 5.19: An  $n$ -body problem with approximate data from the Sun, Venus, Earth and Mars. The right, geocentric view clearly shows the apparent retrograde motion historically observed on Mars.

The implementation of the model of the previous section showed an advanced feature of the ODE editor which helps declare differential equa-

tions for systems of very high dimensions. In the case in which the variables are not very much interrelated, i.e. the rate of a state variable depends only in a few other states, differentiating array elements independently (as we did) can appropriately solve the problem. In other situations, this approach can be rather inefficient and we demonstrate a more efficient approach in this section.

### Exercise 5.18.

The **OscillatorArray** model computes the acceleration of the oscillators using a single custom method named `double[] acc(double[] y)` that takes an array as input and returns an array containing accelerations. Compare the implementation of the custom methods in the **OscillatorChain** and **OscillatorArray** models and also note the very slight difference in the ODE editors. Which custom method is easier to code? Which is easier to understand? Which is more efficient? Give reasons for your answers.  $\square$

Consider a small solar system in which three planets with approximately similar masses orbit a central, more massive star following orbits which are in the same plane. Newton's Universal Gravitation Law states that every two (spherical and with uniform density) bodies attract themselves with a force given by

$$\mathbf{F}_{ij} = -G \frac{M_i M_j}{\|\mathbf{r}_i - \mathbf{r}_j\|^2} \frac{\mathbf{r}_i - \mathbf{r}_j}{\|\mathbf{r}_i - \mathbf{r}_j\|}. \quad (5.8.1)$$

$\mathbf{F}_{ij}$  is the force exerted on the body  $i$  by the body  $j$ .  $G$  is the Universal Gravitational constant  $6.67428 \times 10^{-11} \text{ m}^3 \text{ Kg}^{-1} \text{ s}^{-2}$ , the  $M$ 's are the masses of the bodies, and the  $\mathbf{r}$ 's the position vectors of the centers of the bodies. The minus sign in equation (5.8.1) makes this an attractive force, directed along the relative position vector  $\mathbf{r}_i - \mathbf{r}_j$ .

Implementing the resulting system of equations using an approach similar to that of Section 5.7 is possible but very inefficient. In the first place, notice that computing the acceleration of a given body means computing the sum of the forces exerted by all other bodies. However, because  $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$  (Newton's Third Law), we are bound to compute each force twice. Also, computing separately the  $x$  and  $y$  components of any of these forces will make us repeat the costly computation of  $\|\mathbf{r}_i - \mathbf{r}_j\|$ .

To help us improve the efficiency in cases like this, the ODE editor of *EJS* allows us to specify all the states of the system in a single one-dimensional array, which is then differentiated as a whole. The **SolarSystem** model displayed in Figure 5.19 creates a one-dimensional **state** array with as many as `4*nBodies` entries. Every four entries of this array will contain the values of  $x$ ,  $y$ ,  $v_x$ , and  $v_y$  for each of the bodies. Figure 5.20

shows how the ODE editor implements the differential equation for our solar system using a single differential equation. The special way in which the state has been entered, typing it with the explicit `[]` suffix, tells the editor that the rate will be given also using an array of doubles.

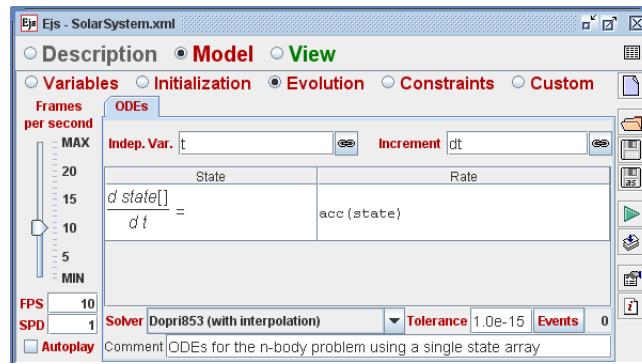


Figure 5.20: The ODE editor using a single state array. Notice the `[]` suffix on the state variable. The rate cell for this state should then return an array of doubles of the same length as the state.

Because we will be displaying the solution of the trajectories every 7 days and there is an important variation of the forces with the distance, we have selected an adaptive solver with a very demanding tolerance of  $10^{-15}$ . The *DoPri853* solver uses a very high order Runge–Kutta adaptive algorithm to reach this tolerance together with interpolation that minimize the number of computations. *DoPri853* is probably the most powerful and efficient algorithm implemented in *EJS*.

The *Auxiliary Vars* page of variables of this model also declares a `rate` array with the same length as `state` and two smaller (of length  $2 * nBodies$ ) arrays called `ZERO` and `force`. These arrays are used to compute the forces on each body using custom methods. The code for the `acc` method which computes the acceleration of each body is

```
public double[] acc (double state[]) {
    computeForces(state);
    for (int i=0,i2=0,i4=0; i<nBodies; i++,i2+=2,i4+=4) {
        rate[i4 ] = state[i4+1]; // x rate is vx
        rate[i4+1] = force[i2]/mass[i]; // vx rate is fx
        rate[i4+2] = state[i4+3]; // y rate is vy
        rate[i4+3] = force[i2+1]/mass[i]; // vy rate is fy
    }
    return rate;
}
```

The `for` loop looks cumbersome but, once the forces are computed,

it just uses the  $(x, vx, y, vy)$  structure of the `state` array to fill the corresponding `rate` entries. The model derived from equation (5.8.1) is really implemented in the `computeForces()` method.

```
public void computeForces(double[] state) {
    System.arraycopy(ZEROS,0,force,0,force.length); // clears the force array
    for(int i=0,i2=0,i4=0; i<nBodies; i++,i2+=2,i4+=4) { // for each body
        for(int j=i+1,j2=2*j,j4=4*j; j<nBodies; j++,j2+=2,j4+=4) { // against other bodies
            double dx = state[i4 ] - state[j4 ];
            double dy = state[i4+2] - state[j4+2];
            double r2=dx*dx+dy*dy;
            double r3=r2*Math.sqrt(r2);
            double fx=G*mass[i]*mass[j]*dx/r3;
            double fy=G*mass[i]*mass[j]*dy/r3;
            force[i2]  -= fx; // force array alternates fx and fy
            force[i2+1] -= fy;
            force[j2]  += fx;
            force[j2+1] += fy;
        }
    }
}
```

Because the second loop starts from  $j=i+1$ , each force is only computed once, using Newton's Third Law to update simultaneously the forces on bodies  $i$  and  $j$ . Notice also the  $x$  and  $y$  components of each force are derived with the same computational effort.

See how we used the `ZEROS` array (which is a constant array of zeros) to clean the contents of the `force` array. The use of the `System.arraycopy` method provided by Java is more efficient than a loop of the form

```
for(int i=0; i<force.length; i++) force[i] = 0.0;
```

The simulation displayed in Figure 5.19 shows the trajectories of a mini solar system composed of the Sun, Venus, the Earth, and Mars using real astronomical data taken from [?]. The data, indicated in an initialization page, is appropriately converted so that the unit of length is one Astronomical Unit ( $1\text{AU} = 149.597 \times 10^9$  meters), the unit of mass is the mass of the Earth ( $5.9736 \times 10^{24}$  Kilograms), and the unit of time one day. A second initialization page computes the center of masses and makes it the (inertial) reference frame. This change of frame is helpful for situations in which there is not a very massive body which won't barely move and can be used as the center of the reference frame.

The view uses the `ParticleSystem` and `TraceSet` elements to display the motion of the bodies both in a frame located in the center of mass



and in a geocentric view. This second view allows us to see the apparent retrograde motion of the planets as seen from the Earth. See Figure 5.19. In order to give the bodies different colors we defined an array of variables of `Object` type and initialized the elements in this array using the Java class `java.awt.Color` (see Appendix ??). The particles are also given different sizes (although not to scale).

**Exercise 5.19.**

The *Solar System* initialization page contains also data for Jupiter and Saturn. Change the value of the `nBodies` variable to 6 to see them. □

**Exercise 5.20.**

This exercise shows that if the Sun wouldn't have been so massive as compared to its planets, the motion of the Earth (and therefore life on it) would have followed a much more different fate. Make Mars much more massive (as much as one-tenth the mass of the Sun) and see how the orbits of the inner planets become erratic. □

**Exercise 5.21.**

Simulate the orbit of a planet orbiting a binary star. A binary star is actually a set of two stars with similar masses orbiting around each other. Make Mars as massive as the Sun and locate Venus and the Earth away from both. Show that orbits close to the binary star are very unstable (eventually even escaping from the system) while those far enough from the center of masses of the binary stars approximate elliptical trajectories.

Disable the *Center of Masses* initialization page (right-click on its tab and select the entry *Enable/Disable this page*) and see how the whole system slowly drifts up making the description of motion even more complicated. Choosing the right reference frame is frequently an important decision. □

## 5.9 DIRECTION FIELDS OF PLANAR AUTONOMOUS SYSTEMS\*

[Note to reader: The 3D Vector Field element used in this section is preliminary. The 3D code here will likely change in the next release of *Easy Java Simulations*.]

The Lotka–Volterra model is an example in which the typical qualitative behavior of all trajectories is more interesting than a particular solution with given initial conditions. Providing initial conditions for the number of different fishes in the Mediterranean Sea, for instance, can only be done through statistical (approximate) means. Because the model is non-linear, we need to use numerical and graphical techniques to study the qualitative behavior of solutions.

It is possible to define a direction field for two-dimensional systems in which a segment is drawn for each point in  $(x, y, t)$ -space, similarly to what we did for one-dimensional systems in Section 5.4. However, the plot will now be three-dimensional and difficult to apprehend. For two-dimensional, autonomous systems of the form

$$\begin{aligned}\dot{x} &= f_1(x, y) \\ \dot{y} &= f_2(x, y),\end{aligned}\tag{5.9.1}$$

it is more common to plot a map similar to a phase-space. We then draw a segment at each point of a grid in the  $(x, y)$ -plane, according to the direction given by the vector  $(f(t, x, y), g(t, x, y))$ . This plot is also called a direction field (although it is different from the direction field introduced previously) because it is the field for the one-dimensional ODE obtained dividing the equations in (5.9.1):

$$\frac{dy}{dx} = \frac{f_2(x, y)}{f_1(x, y)}.\tag{5.9.2}$$

(Notice that in this new problem, the independent variable is now  $x$ .)

The **DirectionFieldPlotter3D** simulation is a variation of **DirectionFieldPlotter** in which we plot the direction field and the solution in both the three-dimensional  $(t, x, y)$  space and in the two-dimensional  $(x, y)$  phase-space. Figure 5.21 shows the plots for the Lotka–Volterra equations for typical values of the parameters.

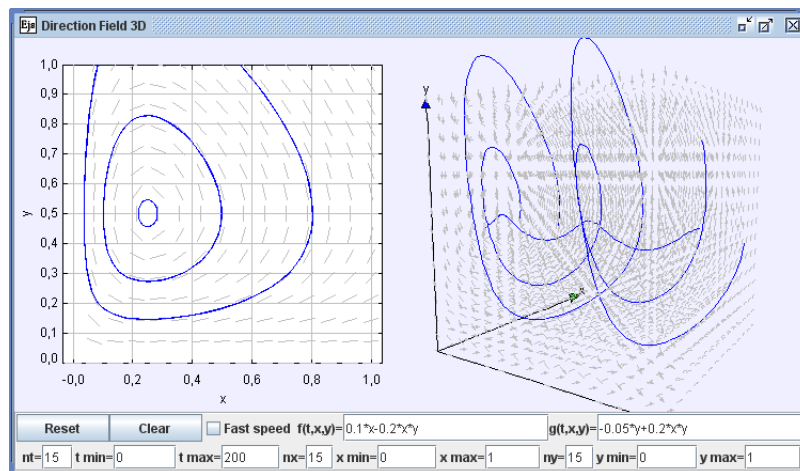


Figure 5.21: The grid of small grey arrows indicate the direction field of the ODE in  $(x, y)$  phase-space (left) and in  $(t, x, y)$  three-dimensional space. The ODE shown is Lotka–Volterra’s model for the predator and prey system.

The implementation of this simulation extends that of **DirectionFieldPlotter** in obvious ways. The main difference is that we allow the user

to select initial values for the populations,  $x_0$  and  $y_0$ , which we take as initial conditions at time  $t_0 = 0$ . Hence, we always compute the solution curve forwards in time. The view of the simulation includes three-dimensional drawing elements and, although they behave in a similar way to two dimensional elements in the plane, we must postpone their discussion until Chapter ???. But we thought it was important to be able to compare the three- and two-dimensional views of the solution curve and the trajectory, respectively.

Notice that, in three-dimensional space, a solution curve never passes twice through the same point because  $t$  is always increasing. Uniqueness of solutions imply also that two solutions never cross. The behavior of trajectories  $(x(t), y(t))$  in phase-space can be different, since trajectories are projections of the three-dimensional solution curves along the time axis. (To experience the feeling of this projection, left-click and drag the three dimensional scene to rotate it until the time axis points towards you.)

### Exercise 5.22.

Use the **DirectionFieldPlotter3D** simulation to plot the fields and trajectories of other two-dimensional systems. For instance, try a modification of the Lotka–Volterra model in which preys must compete among themselves for limited food supply:

$$\begin{aligned}\dot{x} &= ax - bxy - ex^2, \\ \dot{y} &= -cy + dxy.\end{aligned}\tag{5.9.3}$$

Try  $e = 0.1$  for the value of the parameters shown in Figure 5.21 and increase the maximum allowed time to 500. The trajectories in phase-space lose their periodic behavior and spiral now towards an equilibrium point. An equilibrium point existed also in the unmodified Lotka–Volterra model, but now becomes attractive or *asymptotically stable*. This behavior means that, no matter their initial values, in the long run all populations approach a constant value given by the coordinates of the equilibrium point.  $\square$

### Exercise 5.23.

Does it make sense to plot the phase-space direction field for non-autonomous systems? Judge by yourself. Enter the non-autonomous system

$$\begin{aligned}\dot{x} &= 0.05(1.5 + \sin(t/50))x - 0.2xy \\ \dot{y} &= 0.2xy - 0.05y.\end{aligned}\tag{5.9.4}$$

in **DirectionFieldPlotter3D** and see how the field changes with time. Now trajectories in state-space can cross each other but also themselves in a non-periodic form.  $\square$

## PROBLEMS AND PROJECTS

**Project 5.1** (Validation of Newton's proportional cooling law). Before using it extensively, models need to be validated by comparing their predictions to known solutions of the problem or to experimental data. The **Cooling-CoffeeData.txt** file in this chapter's **data** subdirectory contains data from a real experiment of a cooling cup of black and creamed coffee at a constant room temperature. The fit of Figure 5.1 was created with the simulation **CoolingCoffeeValidation**, which reads the data and plots it, together with the graph of the solution given by (5.1.2). Run this simulation, which has initially wrong parameters  $T_r$  and  $k$ , and try to adjust the parameters of this solution so that both plots match to a reasonable degree of accuracy. [Hint: Run the simulation with the default parameters and right-click the experimental time series to bring in a data set tool as indicated in Appendix ??]. Define a new fit according to equation (5.1.2). Use the parameters provided by the **Autofit** checkbox and re-run the model.]

The **CoolingCoffeeValidation** simulation uses the class **ResourceLoader** from the package `org.opensourcephysics.tools`. This class handles access to data typically found on disk files. The class takes care of finding the file and converts the data in it into a ready-to-use format. See Table 5.1.

Table 5.1: Examples of methods in the class **ResourceLoader**.

<code>getString(String path)</code>	Reads the contents of a text file and returns it as a single <code>String</code> . Lines are separated by the new line “\n” character.
<code>getIcon(String path)</code>	Returns an object of the <code>javax.swing.ImageIcon</code> class with the contents of the graphic file.
<code>getIcon(String path)</code>	Returns an object of the <code>java.applet.AudioClip</code> class with the contents of the audio file.

**Project 5.2** (Validation of Newton's proportional cooling law with changing outside temperature). The **CoolingCoffeeEditorValidation** model uses the ODE editor to solve equation (5.3.1) with an outside changing reference temperature provided by data recorded experimentally. The file **Cooling-WaterData.txt** in the **data** directory provides records of the cooling of hot water on an insulated steel cup and a regular cup, together with the surrounding (cooler) temperature for 8 hours.<sup>4</sup> Reading the data file and

<sup>4</sup>The data was kindly recorded for us by Roger Frost (see <http://www.rogerfrost.com>) outdoors in Cambridge, UK, on October 13th, 2007, starting at 9:00 AM.

using it to create a usable `Tr()` method requires some Java technicalities. For instance, we use an object of the class `java.util.ArrayList` to accommodate an unknown number of entry points. We also use a binary search algorithm to interpolate data in between the times provided by the file.

Use the simulation to try to adjust the parameter  $k$  so that the data produced by the theoretical model matches as closely as possible the experimental data. We found the data obtained from the isolated cup provided a better fit than the regular cup, although fitting the first part of the graph is difficult in both cases. What is a possible explanation for this?

**Project 5.3** (Harvesting in the Lotka–Volterra model). To provide a possible explanation of the decrease of percentage of the catch of food fish during the World War I period, Volterra modified the original model (5.5.3) to introduce fishing. A model which uses *constant-effort harvesting* is given by equations

$$\begin{aligned}\dot{x} &= ax - bxy - h_1x, \\ \dot{y} &= -cy + dxy - h_2y.\end{aligned}\tag{5.9.5}$$

for non-negative constants  $h_1$  and  $h_2$ . Modify the models of this chapter to show that a moderate amount of fishing increases the averages of prey and decreases that of predators. This was Volterra’s explanation of the phenomenon, since fishermen were involved in the war during the conflict and harvesting decreased noticeably during those years.

**Project 5.4** (Dumped, driven pendulum). Although we will deal with pendula in detail in Chapter ??, modify the **SimplePendulum** model to introduce a dumping term and a driving force. The resulting equation is

$$\ddot{\theta} = -\frac{g}{L} \sin(\theta) - b\omega + T(t).\tag{5.9.6}$$

The positive parameter  $b$  provides a frictional force proportional to the angular velocity. The forcing term  $T(t)$  corresponds to an external torque on the pendulum. Use a sinusoidal function of the form  $A \sin(ft)$ , where  $A$  is the amplitude of the torque and  $f$  its frequency. Explore how the new terms affect the phase-space portrait.

**Project 5.5** (Length of a planet’s year). Run the **SolarSystem** simulation with an increment of time of one day and compute approximately the length of the year for each of the planets. Notice the difference with the real lengths (224.70 days for Venus, 365.25 for the Earth, and 779.96 for Mars). The reason is that we have used the average orbit velocity for each planet as the velocity at its aphelion (the farthest point from the Sun). Explain why this is incorrect and try to provide a more accurate value of the velocity at the aphelion (and hence of the year length) of the planets.

**Project 5.6** (Logistic growth). (DRAFT TEXT) Use the techniques discussed in this chapter to study the solution of the continuous Logistic Growth model given by  $\dot{P} = r(1 - P)P$ . Add harvesting ( $+H_0 \sin(2\pi t)$ ) and/ or migration to it  $+q(t)$  (Borelli and Coleman pg 127 and 125, resp.).

**Project 5.7** (Trajectories in Dipole Fields). A charged particle in electric and magnetic dipole fields.

**Project 5.8** (Particles with Drag and Spin). Physics of sports.

**Project 5.9** (Airplanes with Lift). Flight.

## APPENDIX: NUMERICAL METHODS

[To be written]