Chapter 2
NetLogo and Physics

William M. Spears

2.1 A Spring in One Dimension

This chapter will introduce you to NetLogo and Newtonian physics. Let's start with NetLogo first. NetLogo is a tool that allows us to simulate distributed, non-centralized systems of agents. As we discussed in Chap. 1, agents are autonomous entities (either in software or hardware) that observe, react and interact with other agents and an environment. People and other living creatures are agents. Robots are agents. Some agents are computer programs that exist only in computer networks. NetLogo allows us to simulate many types of agents. There are two ways you can interact with the NetLogo simulations in this book. The simplest way is to run them as Java applets in your browser. In this case, you will need to make sure that Java 5 (or a later version) is installed on your computer. You will be able to run the simulations in your browser, but you will not be able to modify the code. If you have trouble with one browser, try another. Browsers that work well are Firefox, Internet Explorer, Flock, Chrome, Opera and Safari.

Alternatively, you can run the NetLogo simulations directly on your computer. This is what we will assume throughout this book. In this case, you will have to install NetLogo. We will be using NetLogo 4.1.2. Currently, the NetLogo web page is at http://ccl.northwestern.edu/netlogo/. If this link does not work, use a search engine to locate NetLogo 4.1.2. Then follow the download and installation instructions. NetLogo will run on a Windows machine, a Macintosh or under Linux.[1] When you are done, download the NetLogo code that comes with this book. We are going to start with a spring simulation called "spring1D.nlogo" (nlogo is the suffix used for

William M. Spears
Swarmotics LLC, Laramie, Wyoming, USA, e-mail: wspears@swarmotics.com

[1] The NetLogo code in this book has been tested on Debian and Ubuntu Linux, Mac OS X, as well as Windows Vista and Windows 7.

W.M. Spears, D.F. Spears (eds.), *Physicomimetics*,
DOI 10.1007/978-3-642-22804-9_2,

NetLogo programs). On a Windows or Macintosh, simply download this file and double-click on it. If you are using Linux, run "netlogo.sh". If everything works properly, one window with three tabs should open. One tab is for the NetLogo code (which we will talk about later). Another tab is for the documentation. The third is the graphical window. Select File, then Open..., and select "spring1D.nlogo" (you may have to navigate your way to the file, depending on where you placed it on your computer). Now your graphical window should look like Fig. 2.1.

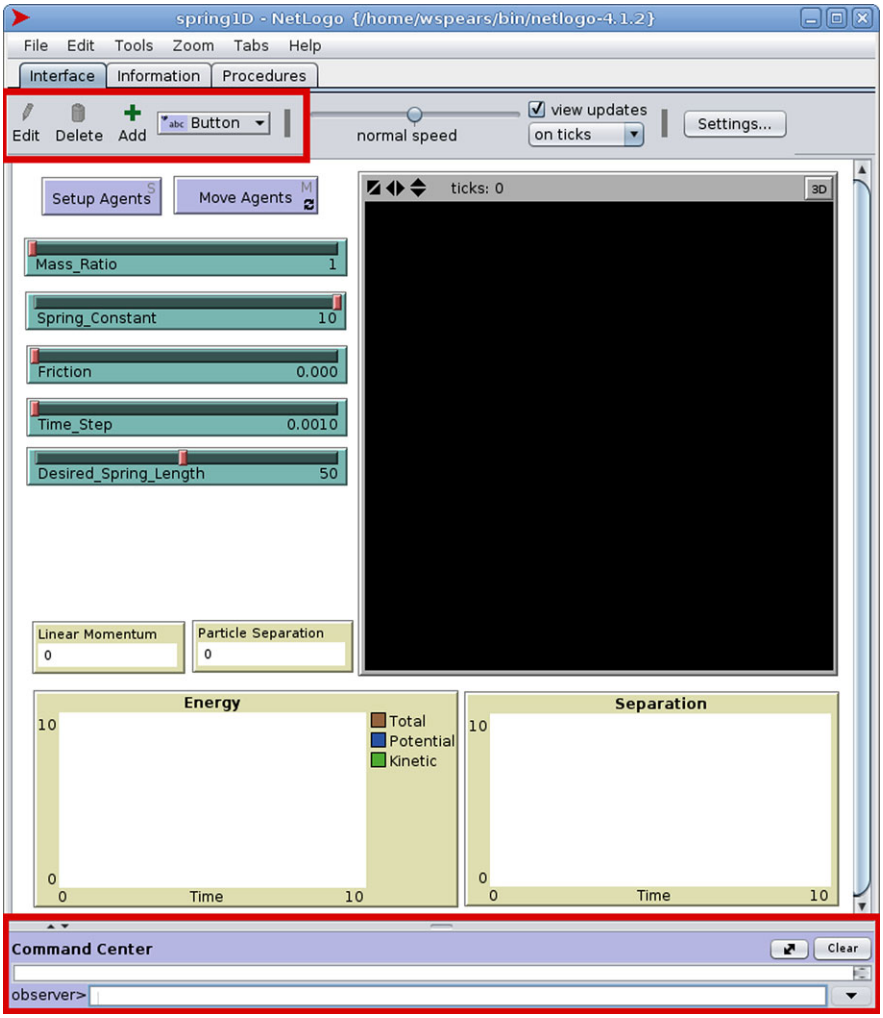

Fig. 2.1: A picture of the NetLogo graphical user interface for the one-dimensional spring simulation

If you are using a web browser, you will click on "spring1D.html" instead. In this case, the graphical window shown in Fig. 2.1 should appear in your browser. As you scroll down you will also see the documentation about this simulation, and the actual code will be shown at the bottom.

A full explanation of NetLogo is beyond the scope of this book. For example, we will not use the upper left portion and bottom portion (highlighted in red) of the graphical window shown in Fig. 2.1. However, we will explain enough to get you started. It is very intuitive, and once you start to learn NetLogo you will gain confidence very quickly. The graphical interface to most of the simulations shown in this book contains several components. Violet buttons allow you to start and stop the simulation. For example, Setup Agents initializes the agents. The other button, Move Agents, tells the agents to start running. If you click it once, the agents start. If you click it again the agents stop. If you look carefully you will notice single capital letters in the buttons. Initially, they are light gray in color. If you mouse click in the white area of the graphical window, the letters will become black. When they become black the letters represent keys you can type on your keyboard to control the simulation. In this case, you can type "S" or "s" to set up the agents (NetLogo is not case-sensitive in this situation). Then you can type "M" or "m" to start and pause the simulation. These action keys are not available, however, if you are running the simulation in your browser.

Green sliders allow you to change the values of parameters (variables) in the simulation. Some sliders can be changed when the simulation is running, while others can be changed only before initialization. The "initialization sliders" and "running sliders" depend on the particular simulation. The light brown bordered boxes are "monitors" that allow you (the observer) to inspect values of parameters as the simulation runs. Similarly, graphs (shown at the bottom) also allow you to monitor what is happening in the simulation. Graphs show you how values change over time, whereas monitors only give you values at the current time step. One other useful feature is the built-in speed slider at the top of the graphical user interface. The default value is normal speed, but you can speed up or slow down the simulation by moving the slider. This slider is available when running the simulation in your browser.

So what precisely is "spring1D"? It is a one-dimensional simulation of a spring that is horizontal. There is a mass connected to each end of the spring. There is a slider called Desired_Spring_Length that represents the length of the spring when no force is applied. It is set to 50 (with NetLogo simulations such as this, units are in terms of screen pixels) in the simulation, but you are free to change that. Consider the following Gedanken (thought) experiment. Imagine that the two masses are pulled apart beyond 50 units (this is not something you are doing with NetLogo yet; this is an experiment you perform in your mind). What happens? The spring pulls the masses closer. On the other hand, if the masses are closer than 50 units, the spring pushes them back out.

Another important parameter that is controlled by a slider is the Spring_Constant. This is simply the stiffness of the spring. If the spring constant is higher, the spring is stiffer. For now we will focus on the Particle Separation monitor and Separation graph. The monitor indicates the distance between the two weights at any moment in time. The graph shows how this separation changes over time.

Let's run the simulation. Start the code, but do not change any parameters. First, click Setup Agents. You will see two small white dots. These represent the weights at the ends of the spring. You will also see a small red dot, which is the center of mass of the spring system. We will explain this more later. For now you will note that the red dot is halfway between the two white dots. This is because we are currently assuming that the two weights at the ends of the spring are of equal mass. Click Setup Agents a couple of times. You will notice that the initial positions change each time. This is because NetLogo uses a random number generator. If you have any randomness in the system, the simulation will run differently each time. In the case of this particular simulation, the only elements of randomness are the initial positions of the two weights. The initial position of the red dot representing the center of mass also changes with each experiment, since it depends on the initial positions of the two weights.

Now click Move Agents and the simulation will start. At any time you can click on it again to pause the system, and then click again to continue. Watch the two white dots. Because of the randomized initialization, the two white dots are highly unlikely to be placed initially such that they are precisely 50 units apart. If they are closer than 50 units, they immediately move away from each other. If they are farther than 50 units, they move towards each other. Then they oscillate back and forth, as we would expect from a spring.

Examine the Particle Separation monitor. You will note that the separation increases and decreases around 50. In fact, the amount of deviation from 50 is the same in both directions, i.e., if the maximum separation is $50+x$, then the minimum separation is $50-x$. You can also see this oscillation clearly on the Separation graph. An example is shown in Fig. 2.2. The x (horizontal) axis represents time. The y (vertical) axis represents the amount of separation between the weights on the ends of the spring.

Feel free to play with the system. Stop everything, change the desired spring length, and rerun the system. Similarly, decrease the spring constant (to one, for example). What happens? Precisely what you would expect. The spring is weaker and less stiff, so the oscillations are longer. In fact, change the desired spring length and/or spring constant while the system is running! This is analogous to modern automotive suspension systems that can be adjusted by the driver—pushing a switch inside a car changes the characteristics of the spring. Very soon you will understand the dynamics of a spring.

To continue with the automotive analogy, the movement of each wheel is damped by a spring connected to the car. But the wheel weighs much less than the car. This is essentially the Mass_Ratio slider in the simulation. The

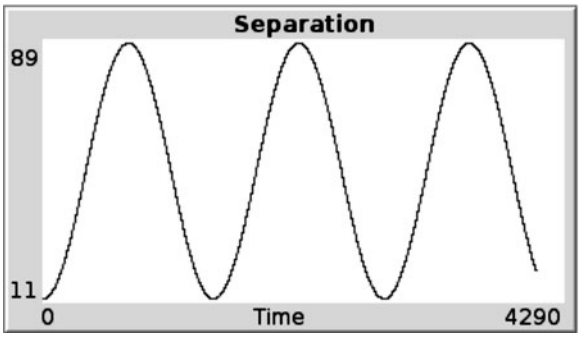

Fig. 2.2: A graph of the length of the spring over time in the one-dimensional spring simulation

mass on one end of the spring has a fixed weight of one. However, during initialization (and only then) we can adjust the weight of the other mass to be one or greater. For example, set it to 10 and rerun the system. What you will notice is that one dot hardly moves, while the other moves a lot. The dot that is almost stationary is the heavier mass. Simply put, the car doesn't move much while the wheel does.

You will also notice that the red dot representing the center of mass of the system is no longer halfway between the two white masses, but is very close to the "heavy" dot. Although we will discuss this more precisely later in the chapter, it is the location where the system is balanced. If you replaced the spring with a bar, and attached the two masses to the ends, the center of mass is where you would pick up the bar so that it would be balanced properly.

At this point you will have noticed that the system continues to oscillate forever. This is an "ideal" spring that doesn't exist in the real world. Real springs are damped by friction and eventually come to rest (assuming they are not perturbed again). And in fact, this is precisely what we would want a car spring to do. We do not want our car to bounce forever because of one pothole. Shock absorbers provide friction that damp the spring. We can simulate this easily by changing the Friction slider from zero to 0.001. First, start the simulation with zero friction, and let it oscillate for a while. Then increase the friction to 0.001. Very quickly the oscillations will damp and the separation between the two masses will settle at the desired separation distance again. An example is shown in Fig. 2.3.

Up to this point we have not discussed the Time_Step slider, the Linear Momentum monitor or the Energy graph. These are somewhat more technical and require a deeper discussion of Newtonian physics. As we discuss these topics next we will also take the opportunity to indicate how these concepts are programmed in NetLogo.

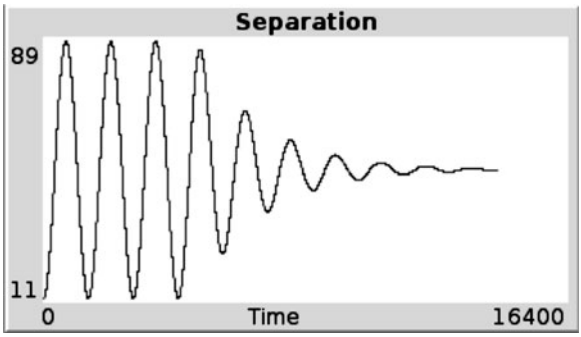

Fig. 2.3: A graph of the length of the spring over time in the one-dimensional spring simulation, when friction is applied after 5,000 time units

2.1.1 NetLogo Details and Newtonian Physics

In addition to the graphical Interface tab, NetLogo has a Procedures tab for the code. NetLogo assumes that there are two *conceptual* perspectives for the code, the agent perspective and the observer perspective. The agent perspective is from the agent's point of view. For historic reasons agents were called "turtles" a long time ago, and this convention still holds. We will use the terms "turtles," "agents," "particles," and "robots" synonymously throughout this book, because in all cases we are discussing a set of distributed cooperating entities. The observer perspective is a more global perspective and generally deals with higher-level concepts. Stated another way, the observer perspective is used to monitor the emergent behavior of the system, while the turtle perspective deals with the individual agents themselves. The distinction between the two perspectives is not always clear, and NetLogo itself does not care much about the difference. But it allows you, the programmer, to more easily maintain different perspectives on the system.

Let's look at the code in Listing 2.1. At the top of the code is a list of `global` variables. These are variables that we (the observer) are interested in examining or controlling. Some of the variables refer to the total linear momentum, total kinetic energy, and total potential energy of the whole system. If you are unfamiliar with these concepts, do not worry. We will define them shortly. Other variables refer to the desired separation `D` between the two masses at the ends of the spring, the spring constant `k` and the location of the center of mass.

In NetLogo, procedures start with the keyword `to` and conclude with the keyword `end`. The glossaries at the beginning of this book summarize the syntax of NetLogo and give a list of important variables used in our NetLogo simulations. The procedure `setup` is called when you click on the Setup Agents button. To see this more clearly, start the NetLogo simulation.

Listing 2.1: A portion of the observer code for the one-dimensional spring

```
globals [total_lm total_ke total_pe total_energy
         center_of_mass_x k FR DeltaT D S]

to setup
   clear-all crt 2
   ask turtles [setup-turtles]
   set S abs(([xcor] of turtle 1) - ([xcor] of turtle 0))

   set center_of_mass_x
      (([mass] of turtle 0) * ([xcor] of turtle 0) +
       ([mass] of turtle 1) * ([xcor] of turtle 1)) /
      (([mass] of turtle 0) + ([mass] of turtle 1))
   ask patch (round center_of_mass_x) 0
      [ask patches in-radius 4 [set pcolor red]]
end
```

Then "select" the Setup Agents button. This can be done by dragging a rectangle around it with your mouse. If it has been selected properly a gray rectangle is displayed around the button. You can resize the button by dragging the small black squares. You can also "right-click" with your mouse and select Edit... with your mouse. A new window is opened, describing what the button does. In this case, it calls the `setup` procedure of the code.[2] Sliders, monitors and graphs can be selected and edited in the same manner, showing you what is being controlled, monitored, and displayed.

Now let's examine what the `setup` procedure does. First, the `clear-all` command clears everything. Then `crt 2` creates two turtles. They are given unique identifiers, namely 0 and 1. Then the turtles are asked to run their own `setup-turtles` routine. The separation `S` between the two turtles is computed with `set S abs(([xcor] of turtle 1) - ([xcor] of turtle 0))`. At this point the procedure computes the center of mass of the system. Mathematically, in a one-dimensional system, the center of mass is:

$$\frac{\sum_i m_i x_i}{\sum_i m_i} \,, \tag{2.1}$$

where the sum is taken over all particles in the system. Since there are only two in this particular simulation, `set center_of_mass_x (([mass] of turtle 0) * ([xcor] of turtle 0) + ([mass] of turtle 1) *`

[2] NetLogo states that you should use control-click rather than right-click if using a Macintosh. See http://ccl.northwestern.edu/netlogo/requirements.html for more information.

Listing 2.2: A portion of the turtle code for the one-dimensional spring

```
turtles-own [hood deltax r F v dv mass ke lm]

to setup-turtles
   set color white home
   set shape "circle" set size 5

   ifelse (who = 0)
     [set mass 1 set heading 90 fd (1 + random (D / 2))]
     [set mass Mass_Ratio set heading 270 fd (1 + random (D / 2))]
end
```

`([xcor] of turtle 1)) / (([mass] of turtle 0) + ([mass] of turtle 1))` computes the center of mass.

Finally, the center of mass is drawn on the graphics screen. In addition to turtles, NetLogo includes the concept of "patches." Patches represent pieces of the environment that the turtles inhabit. For example, each patch could represent grass for rabbit agents to munch on. NetLogo was modeled on the premise that much of the interaction between agents occurs indirectly through the environment, a process referred to as "stigmergy." Stigmergy is an excellent way to model various biological and ethological systems, and hence biomimetics relies heavily on this form of communication. Physicomimetics generally does not make use of stigmergy, because of the numerous difficulties in reliably changing the environment when performing research in swarm robotics. The spring simulation does not rely on the center of mass computation—it is merely a useful mechanism for us (the observer) to understand the dynamics of the simulation. Hence, the final step in the `setup` procedure is to make the patch red at the coordinates of the center of mass by saying `ask patch (round center_of_mass_x) 0 [ask patches in-radius 4 [set pcolor red]]`.

The code for the turtles is shown in Listing 2.2. At the top of the code is a list of "turtle" variables. These are variables that each turtle maintains. They can also be shared with other turtles. Each turtle knows its neighbors, called the `hood`. In this case, since there are only two turtles, each turtle has only one neighbor. Each turtle also knows the x-displacement to the neighbor (which can be positive or negative), the range to the neighbor, the force felt on itself, its velocity, change in velocity, mass, kinetic energy and linear momentum. We will discuss these concepts further below.

Both turtles execute the `setup-turtles` procedure. First, each turtle is given a white color. Then they are placed at the `home` position, which is at the center of the graphics pane, and are drawn as large circles. Each turtle

Listing 2.3: Remainder of observer code for the one-dimensional spring

```
to run-and-monitor
   ask turtles [ap]
   ask turtles [move]

   set center_of_mass_x
      (([mass] of turtle 0) * ([xcor] of turtle 0) +
       ([mass] of turtle 1) * ([xcor] of turtle 1)) /
      (([mass] of turtle 0) + ([mass] of turtle 1))
   ask patch (round center_of_mass_x) 0
      [ask patches in-radius 4 [set pcolor red]]

   set total_lm sum [lm] of turtles
   set total_ke sum [ke] of turtles
   set total_pe (k * (S - D) * (S - D) / 2)
   set total_energy (total_ke + total_pe)
   tick
end
```

is identified by a unique identifier given by `who`. One turtle is given a mass of one by default, is turned 90° clockwise to face to the right (in NetLogo a heading of 0° points up), and moves randomly forward an amount determined by the desired spring length `D`.[3] The other turtle is given a mass determined by the Mass_Ratio slider, is turned 270° clockwise to face to the left, and moves randomly forward. The effect is to create a horizontal spring with a random spacing.

As stated before, after initialization, you start the simulation by clicking on the Move Agents button. If you "open" the button using the instructions given earlier (select the button and then click on Edit...) you will see that the Observer procedure `run-and-monitor` is executed. Note that the Forever box is checked, indicating that the simulation will run forever, unless you pause it by clicking the Move Agents button again.

The `run-and-monitor` procedure is shown in Listing 2.3. This procedure, which is looped infinitely, performs numerous actions. First, it tells each turtle to execute a procedure called `ap`. This calculates where each turtle should move. Then each turtle is asked to `move`. The center of mass is recomputed and redrawn, to see if *it* has moved. At this point the global variables store the total linear momentum, total kinetic energy, total potential energy, and total energy of the system. Finally, `tick` is called, incrementing the system counter by one. The energy variables are displayed on the Energy graph in the simulation. However, fully understanding these concepts means

[3] The discrepancy between Cartesian and NetLogo coordinates will arise more than once in this book—code that is wrong from a Cartesian standpoint is correct for NetLogo.

Listing 2.4: Remainder of turtle code for the one-dimensional spring

```
to ap                                          ; Artificial Physics!
   set v (1 - FR) * v

   set hood [who] of other turtles
   foreach hood [
      set deltax (([xcor] of turtle ?) - xcor)
      set r abs(deltax)
      set S r
      ifelse (deltax > 0)
         [set F (k * (r - D))]
         [set F (k * (D - r))]
   ]
   set dv DeltaT * (F / mass)
   set v  (v + dv)
   set deltax DeltaT * v
end

to move
   set xcor (xcor + deltax)
   set ke (v * v * mass / 2)
   set lm (mass * v)
end
```

we need to carefully examine the core procedures `ap` and `move`, as shown in Listing 2.4.

The first thing that procedure `ap` does is apply friction. The parameter `FR` is controlled by the Friction slider. If it is set to zero then the velocity `v` does not change. Otherwise the velocity is decreased. It should be pointed out that friction is quite difficult to model properly (i.e., in accordance with real-world observations). In fact, there are numerous forms of friction, and they are not well understood. Because we are less interested in perfect fidelity we are free to use a form that provides behavior that is quite reasonable, as shown in Fig. 2.3.

Next the turtle looks for its neighbors. Since there are only two turtles in this system, each turtle has one neighbor. The turtle sees whether the neighbor is to the left or right of itself. If the neighbor is to the right, then the force on the turtle is computed with `set F (k * (r - D))`. If they are separated by a distance greater than `D`, the turtle feels a positive force to the right. Otherwise it feels a force to the left. If the neighbor is to the left, then the force on the turtle is computed with `set F (k * (D - r))`. If they are separated by a distance greater than `D`, the turtle feels a force to the left. Otherwise it feels a force to the right. Note that this obeys Newton's third law, which states that the reaction between two bodies is equal and

opposite. This "force law" is referred to as Hooke's law, after Robert Hooke, a 17th century British physicist.

Each turtle obeys Newton's second law, namely, $\boldsymbol{F} = m\boldsymbol{a}$. This is stated in vector notation. However, since this simulation is in one dimension we can write it simply as $F = ma$ or $F/m = a$. What does this mean? It means that a force of magnitude F imparts an acceleration a to an object. Note that as the mass m of the object is increased, the same amount of applied force results in decreased acceleration.

What precisely is the acceleration a? Mathematically it is equivalent to $\mathrm{d}v/\mathrm{d}t$. This is the instantaneous change in velocity. However, we must always keep in mind that we wish to implement our equations on computer hardware. Nothing is instantaneous in this domain—everything takes some amount of time. Hence we can approximate a with the discrete time equation $\Delta v/\Delta t$ (the change in velocity over some short interval of time). As Δt approaches zero we get closer to the continuous form $\mathrm{d}v/\mathrm{d}t$. Similarly, velocity v itself can be expressed as the instantaneous change in position $\mathrm{d}x/\mathrm{d}t$, with the approximation $\Delta x/\Delta t$. Hence $\Delta x = v\Delta t$.

Let us write Newton's second law as $F = m\Delta v/\Delta t$. We can reorder the terms so that $\Delta v = \Delta t F/m$. The following NetLogo code computes the change in velocity: `set dv DeltaT * (F / mass)`. Then the new velocity is established with `set v (v + dv)`. This leads to the change in position, computed with `set deltax DeltaT * v`, and finally the position of the turtle is updated in the `move` procedure with `setxcor (xcor + deltax)`. In the simulation Δt is represented with `DeltaT`. The variable `DeltaT` is controlled by the Time_Step slider. Try increasing the Time_Step. You will notice that the simulation runs much faster. However, the behavior is similar (Fig. 2.4). This indicates that this approach can work even if our computer hardware has delays in processing.

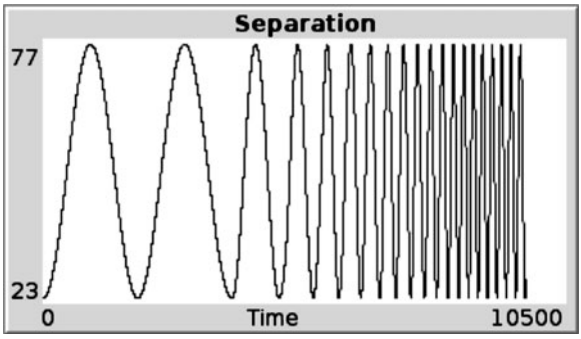

Fig. 2.4: A graph of the length of the spring, where Δt is increased

2.1.2 Conservation Laws

A nice aspect of using physics for agent control is that the emergent system obeys various laws. This serves as a mechanism for verifying that the agent algorithms are working correctly, because if the laws are broken, something is wrong with our algorithms. Also, as we will see later, we can use our understanding of physics to appropriately set some parameters a priori! This is in contrast to other techniques where it is hard to verify that the system is indeed correct, because there are no known laws governing the emergent behavior.

The first example is conservation of linear momentum, which means that the velocity of the center of mass of a system will not change in a "closed system." A closed system is one that experiences no external forces. Our one-dimensional spring is a "closed system" (when we do not modify any sliders). Due to this, the total linear momentum of the system should not change. The linear momentum p of each turtle is $p = mv$. The total linear momentum is simply the sum of the linear momentum of both particles. In Listing 2.4 we see that each turtle computes its own linear momentum as `set lm (mass * v)`. Then Listing 2.3 shows that the total linear momentum is calculated with `set total_lm sum [lm] of turtles`. Because our two turtles are initialized with zero velocity, the total linear momentum should always be zero. Reset the Time_Step to 0.001 and run the simulation again. The Linear Momentum monitor should remain at zero. The fact that it does so indicates that, with respect to this aspect at least, the algorithm is written correctly. The result is a direct consequence of the fact that we are obeying Newton's third law. Another way to state this is that the system should not move as a whole in any direction across the screen. This is indeed what we observe.

Another important law is the conservation of energy. Again, in a closed system, the total amount of energy should remain the same. We focus on two forms of energy, the potential energy and the kinetic energy. What is potential energy? It reflects the ability (potential) to do work. Work is the application of a force over some distance. A spring that is at the desired spring length does not have the potential to do any work because it cannot exert any force. However, a spring that has been expanded or compressed does have the potential to do work. The potential energy of a spring is computed by the observer with `set total_pe (k * (S - D) * (S - D) / 2)`.[4] Kinetic energy, on the other hand, reflects both the mass and velocity of the two turtles. Each has kinetic energy $mv^2/2$, calculated by each turtle with `set ke (v * v * mass / 2)`. The total kinetic energy is computed by the observer with `set total_ke sum [ke] of turtles`. In the absence of friction, the sum of the potential energy and kinetic en-

[4] Let $s = S - D$. Then the potential energy is $-\int_0^s -k\,x\,\mathrm{d}x = k\,s^2/2$.

ergy should remain constant. We compute this in the observer with `set total_energy (total_ke + total_pe)`.

This does not say that both the potential energy and kinetic energy remain constant, merely that the sum remains constant. In fact, the amount of both forms of energy change with time. Rerun the simulation again and note the Energy graph (Fig. 2.5 provides an example). Again the x (horizontal) axis represents time. The y (vertical) axis represents the amount of kinetic energy (green curve), potential energy (blue curve) and total energy (brown curve). What behavior do we see? There is a constant trade-off between kinetic energy and potential energy. However, the total remains constant (be sure to set the Time_Step to 0.001 and Friction to zero). The fact that the total energy remains quite constant provides more evidence that our simulation is correct.

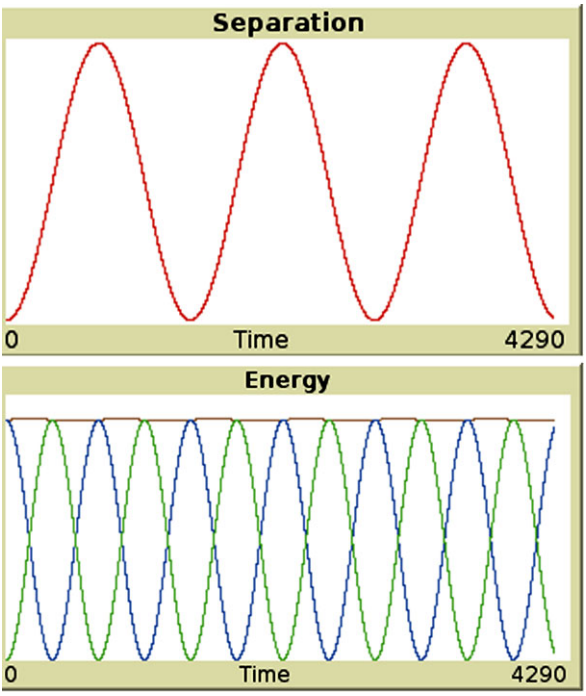

Fig. 2.5: A graph of the kinetic, potential and total energy of the spring, over time, when there is no friction

You will also notice something that you probably noticed before—the turtles appear to move slowly when the spring is maximally compressed or maximally expanded, while moving most quickly when they are `D` apart. The explanation for this can be seen in the Energy graph. When maximally compressed or expanded the potential energy is maximized. Hence the kinetic energy is minimized and the turtles move slowly. However, when they are `D`

apart, potential energy is minimized and kinetic energy is maximized. The correlation between the energy graph and the separation graph is shown in Fig. 2.5.

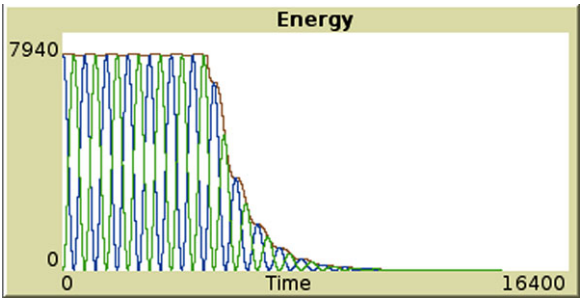

Fig. 2.6: A graph of the kinetic, potential and total energy of the spring, over time, when friction is introduced

However, increase the Time_Step. You will see small fluctuations in the total energy. This is because the conservation laws assume continuous time rather than discrete time. As our simulation becomes more "coarse" from a time point of view, it can slightly violate the conservation laws. However, what is important to note is that (thus far), the errors are really not that large. The system still almost fully obeys the conservation of energy law. Hence, when we use actual computer hardware (such as robots), we will find that the physics laws still apply.

It is instructive to reset the Time_Step to 0.001, but raise Friction. What happens? The total energy of the system gradually disappears, as shown in Fig. 2.6. This is not a violation of the conservation of energy law. Instead it reflects that the NetLogo simulation is not measuring frictional energy (which in real-world systems is generally heat). What is happening is that the total energy of the original system is being converted to heat, in the same way that a shock absorber will become hotter when it damps a spring.

There are other ways you can change the total energy of the system. Two examples are by changing the Desired_Spring_Length or the Spring_Constant. In this case, the total energy can increase or decrease. Is this a violation of the conservation of energy? No, because changing the spring length or spring constant requires the observer to "open" the system and modify the spring itself. Hence, when a car adjusts the suspension system, it must do so by adding or removing energy to that suspension system. However, do note that once the change has been made, the total energy remains constant again, albeit at a different level.

Finally, due to the nature of NetLogo itself, there is one additional way to obtain unusual results. The graphics pane in NetLogo is toroidal. This means that when an agent moves off the pane to the north (south), it re-

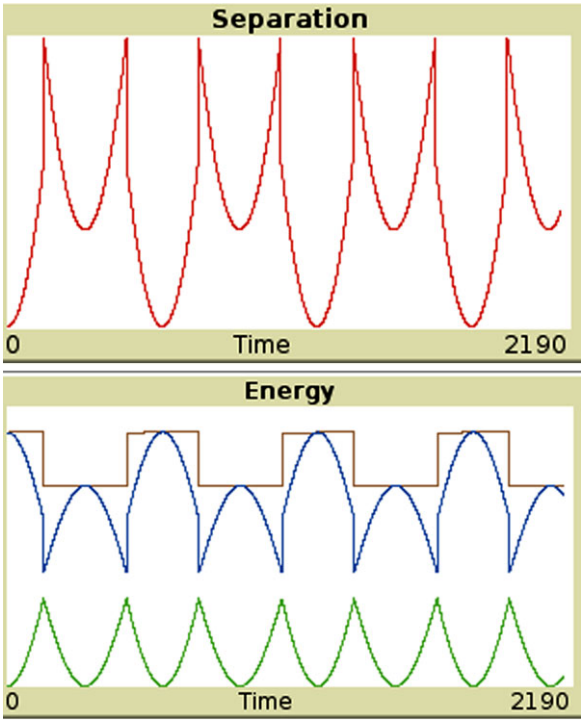

Fig. 2.7: Separation and energy graphs for the one-dimensional spring simulation when the particles cross the world boundary

enters from the south (north). Similarly when an agent moves off the pane to the east (west), it re-enters from the west (east). Newtonian physics assumes Euclidean geometry. The toroidal nature of the NetLogo world breaks the Euclidean geometry assumption. To see an example of the results, open the Desired_Spring_Length slider. Set the maximum value to 500 and the current value to 500. Then run the simulation. Figure 2.7 illustrates some example results for both the Separation and Energy graphs. The Separation graph is odd because when the particle leaves one side of the world, it re-enters from the other, changing the separation dramatically. The Energy graph reflects two different levels of total energy, depending on which side the particle is on. The results are understandable, but do not reflect standard physics. In the real non-toroidal world, with robots, this will never be a problem. But we do need to be careful when using NetLogo for physics simulations to ensure that particles do not cross the environment boundaries.

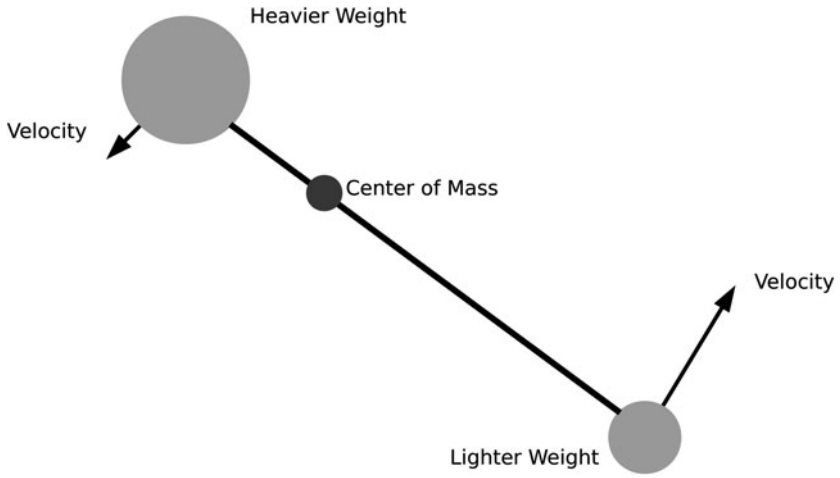

Fig. 2.8: A pictorial representation of a spring that rotates. The light gray circles represent the weights at the ends of the spring, while the dark gray circle represents the center of mass

2.2 A Spring in Two Dimensions

Although the one-dimensional simulation shown above is excellent for a tutorial, we do not expect our robots to be limited to one dimension. Hence, in this section we generalize the simulation to two dimensions (the file is "spring2D.nlogo"). We do this by imparting a rotation to the spring. This is shown in Fig. 2.8. This figure represents the spring with a black line (which can expand or contract). The light gray circles represent the masses on the ends of the spring. The upper mass has greater weight than the lower mass, so the center of mass (a dark gray circle) is closer to the upper mass. At the beginning of the simulation the spring is given an initial rotation around the center of mass. In the case of Fig. 2.8 that rotation is counterclockwise. Note that, as expected, the lower mass moves more quickly (the velocity arrow is longer) because it is farther from the center of mass.

Generalizing the one-dimensional simulation to two dimensions is not very difficult. A picture of the graphical interface is shown in Fig. 2.9. There is one additional monitor for the linear momentum, as both the x- and y-components need to be monitored now. There is one additional slider, called Angular_Motion, representing the amount of initial rotation. Finally, there is a monitor for the angular momentum.

For the two-dimensional spring, the core code is again contained in two procedures, `ap` and `move`. The procedure `ap` calculates where each particle should move, as shown in Listing 2.5. It is essentially the same as in List-

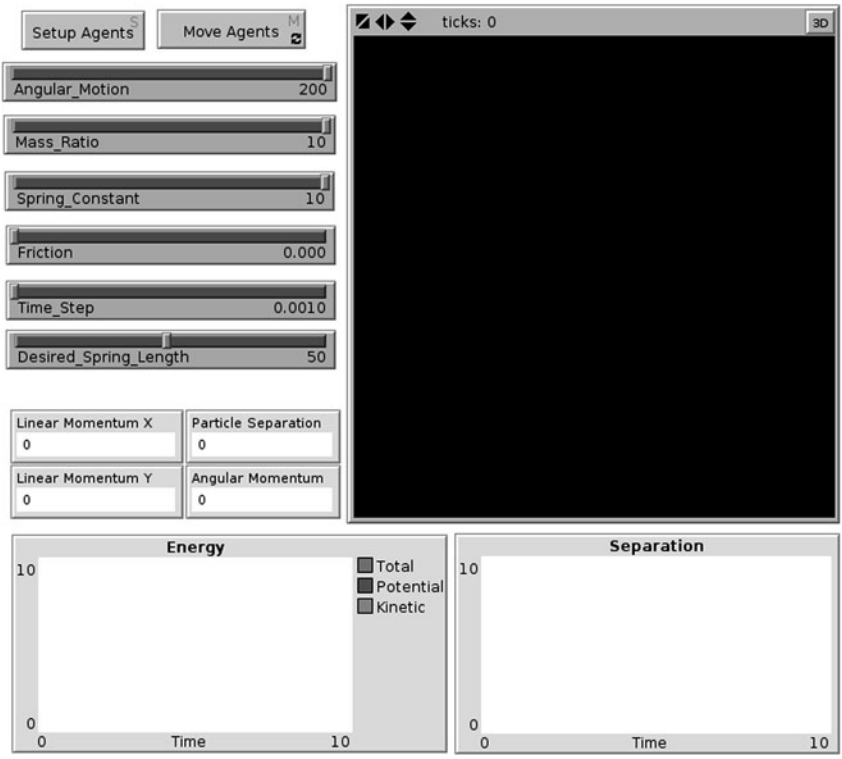

Fig. 2.9: A picture of the graphical user interface for the two-dimensional spring simulation

ing 2.4, except that all operations need to be performed on both the x- and y-components of the system. This is a nice aspect of physics. Motion in two dimensions can be computed by decomposing two-dimensional vectors into their x- and y-components and performing the computations at that level. So, as an example, instead of one line of code to describe how friction changes velocity, we now have two lines of code. There is one line for the x-component of velocity `vx` and one line for the y-component of velocity `vy`.

The procedure `move` (shown in Listing 2.6) moves each particle and computes information pertaining to the momenta and energy. The only large change to the code is at the bottom, where the angular momentum is computed. Angular momentum is the amount of resistance to changing the rate of rotation of a system. The equation is given as $\boldsymbol{L} = \boldsymbol{r} \times \boldsymbol{p}$, where $\boldsymbol{L}$ is the angular momentum vector, $\boldsymbol{r}$ is the position vector of the particle with respect to the origin (in our case this is just the center of mass), $\boldsymbol{p}$ is the linear momentum vector, and $\times$ represents the cross product. In two dimensions this equation can be written with scalars instead of vectors to obtain

Listing 2.5: The main turtle computation code for the two-dimensional spring

```
to ap
   set vx (1 - FR) * vx
   set vy (1 - FR) * vy

   set hood [who] of other turtles
   foreach hood [
      set deltax (([xcor] of turtle ?) - xcor)
      set deltay (([ycor] of turtle ?) - ycor)
      set r sqrt (deltax * deltax + deltay * deltay)
      set S r
      set F (k * (r - D))
      set Fx (F * (deltax / r))
      set Fy (F * (deltay / r))
   ]
   set dvx DeltaT * (Fx / mass)
   set dvy DeltaT * (Fy / mass)
   set vx  (vx + dvx)
   set vy  (vy + dvy)
   set deltax DeltaT * vx
   set deltay DeltaT * vy
end
```

Listing 2.6: The main turtle movement and monitor code for the two-dimensional spring

```
to move
   set xcor (xcor + deltax)
   set ycor (ycor + deltay)

   set lmx (mass * vx)
   set lmy (mass * vy)

   set v sqrt (vx * vx + vy * vy)
   set ke (v * v * mass / 2)
   set lever_arm_x (xcor - center_of_mass_x)
   set lever_arm_y (ycor - center_of_mass_y)
   set lever_arm_r sqrt (lever_arm_x * lever_arm_x +
                         lever_arm_y * lever_arm_y)
   set theta (atan (mass * vy) (mass * vx)) -
             (atan lever_arm_y lever_arm_x)
   set angular_mom (lever_arm_r * mass * v * (sin theta))
end
```

Listing 2.7: The portion of code that imparts the spin to the spring

```
ifelse (who = 0)
   [set mass 1 set vx Angular_Motion]
   [set mass Mass_Ratio
    set vx (- Angular_Motion) / Mass_Ratio]
```

$L = r\,m\,v\,\sin\theta$, where L is the magnitude of $\boldsymbol{L}$ and r is the magnitude of $\boldsymbol{r}$. In our simulation each mass has a velocity. Each velocity can be decomposed into two quantities, the velocity that is along the line of the spring, and the velocity that is at a right angle to the ends of the spring. The former quantity does not impart spin to the system, so we focus only on the latter. This quantity is computed as $v\,\sin\theta$, where θ is the angle of the particle velocity in comparison to the line of the spring. The magnitude of the vector $\boldsymbol{p}$ that is at a right angle to the spring is given by $m\,v\,\sin\theta$. For example, if the particle was moving directly outwards along the line of the spring, $\theta = 0$. Then $\sin\theta = 0$ and $m\,v\,\sin\theta = 0$. Hence $L = 0$ for this particle, because it is not rotating the system. However, if $\theta = 90°$, then $\sin\theta = 1$, $m\,v\,\sin\theta = m\,v$, and $L = r\,m\,v$. This represents a particle that is rotating the system as much as possible.

It is useful to examine $L = r\,m\,v$ more closely. Angular momentum depends on the weight of the masses at the ends of the spring, the distance r from the center of mass, and the velocity. Also, just as with linear momentum, a closed system obeys the conservation of angular momentum! This explains something we have all observed. When an ice skater is spinning, and then brings in her arms, she spins much faster. This is because r has decreased. Since her mass m is constant, v must increase. For example, if r is decreased to one-third of the initial value, velocity must increase three-fold.

The Angular_Motion slider controls the initial amount of rotation, as shown in Listing 2.7. For the lighter particle `vx` is set to `Angular_Motion`. To ensure that both masses of the spring are initialized properly, the x-component of the velocity of the heavier particle is `-Angular_Motion / Mass_Ratio`, where `Mass_Ratio` is set by the Mass_Ratio slider. This creates a system with zero linear momentum.

Run the simulation with the default parameter settings. You will notice that the spring rotates around the center of mass, while expanding and contracting. The Linear Momentum monitors remain at zero, indicating that the center of mass is not moving. The Angular Momentum monitor also stays constant (or very close to constant—a small amount of error can occur due to numerical precision issues in the simulation). Again, this serves to verify that

the simulation is indeed written correctly. By convention a negative value for the angular momentum indicates a clockwise rotation. A positive value indicates a counterclockwise rotation.

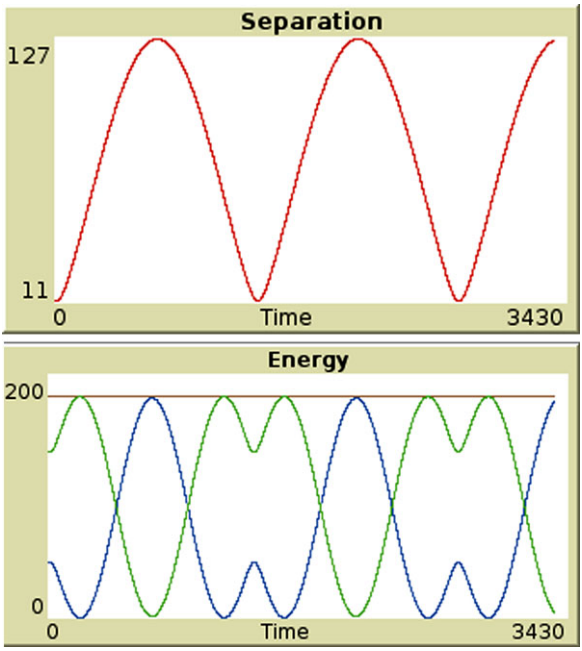

Fig. 2.10: Separation and energy graphs for the two-dimensional spring simulation

Figure 2.10 shows both the Separation graph and the Energy graph. Although still quite cyclic, the Separation graph shows two new behaviors. First, the separation can exceed twice the desired spring length (which is 50). Also, one can clearly see that the average separation is greater than the desired spring length. What is happening? We are seeing the effects of "centrifugal force" due to the rotation. Centrifugal force is an outward force from the center of rotation. The Energy graph is also more interesting than in the one-dimensional simulation, again due to the effects of the rotation of the system. Although the graph shows cycles, they are more complex than the simple sinusoidal cycles shown earlier.

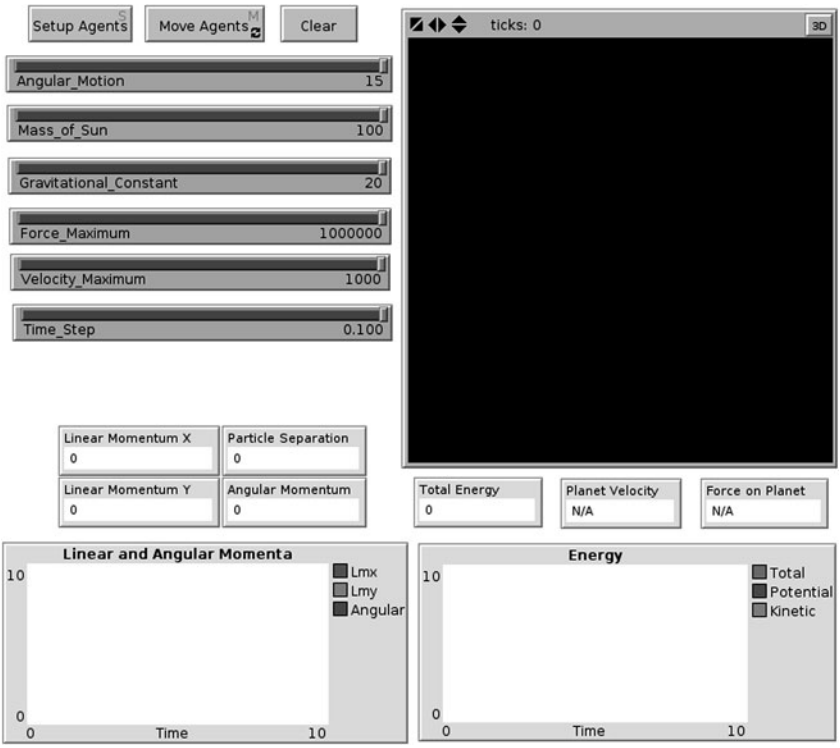

Fig. 2.11: A picture of the graphical user interface for the solar system simulation

2.3 A Simple Solar System

The two-dimensional spring looks quite a bit like a solar system, with the lighter particle being a planet, and the heavier particle being the sun. However, solar system dynamics are governed by Newton's gravitational force law $F = G\,m_1\,m_2/r^2$ (where G is the gravitational constant), not by Hooke's law. The nice aspect of our simulation is that it is simple to modify the simulation to be more like a one-planet solar system, as shown in Listing 2.8. However, note that this particular force law has a feature that Hooke's law does not—the magnitude of the force goes to infinity as r goes to zero. This section will explore how this feature can be problematic for both simulations and real robots, and how the problems can be overcome.

Figure 2.11 shows the graphical user interface for the solar system model, which looks very similar to those we have seen before ("solar_system.nlogo"). The gravitational constant G is controlled by the Gravitational_Constant slider. The mass of the sun, shown as a yellow dot, is controlled with the Mass_of_Sun

Listing 2.8: The portion of code that implements the Newtonian gravitational force law in the solar system model

```
set F (G * mass * ([mass] of turtle ?) / (r * r))
```

slider. The sun is initialized to be at the center of the graphics pane, at $(0, 0)$. The planet has a mass of one and is shown as a white dot. It is initialized to be at location $(15, 0)$, just to the right of the sun. Once again the center of mass is shown as a red dot (and is generally obscured by the sun, since the sun is drawn with a bigger circle). Two new sliders called Force_Maximum and Velocity_Maximum have been introduced, and we will discuss their purpose further down. We have also made use of the `pd` (pen down) command in NetLogo, which is applied to the planet. This draws the orbit of the planet in white. A button called Clear erases the graphics, but allows the simulation to continue. As before, we have an Energy graph, showing the system kinetic energy, potential energy and total energy.[5] Also, monitors showing the total energy, velocity of the planet, and force exerted on the planet (by the sun) have been added. Finally, the Linear and Angular Momenta graph allows us to see if the momenta are changing with time. Blue is used for angular momentum, red for the x-component of linear momentum, and green for the y-component of linear momentum. Recall from before that in a properly designed system, the laws of conservation of both momenta and energy should hold.

The slider Angular_Motion once again controls the amount of rotation of the system. Specifically, the planet is initialized with an upward velocity of `Angular_Motion`, while the sun is initialized with a downward velocity of `-Angular_Motion / Mass_of_Sun`. The result of this is to create a rotating system that has no linear momentum. Unlike the previous simulations, there is no random component in the code. This maximizes the likelihood that the simulation will run similarly for you, making it easier to discuss the behavior and results. For example, let's show the orbits when `Angular_Motion` is 15 and 10. Let's also change the mass of the sun, using the values of 100 and 95. The resulting orbits are shown in Fig. 2.12. The higher initial velocity given when `Angular_Motion` is 15 results in a large elliptical orbit. The orbit is smaller when `Angular_Motion` is 10. The effect of mass on the orbits is qualitatively the same for both values of `Angular_Motion`. As mass decreases the orbit increases in size. This is because the gravitational force is weaker. The effect is bigger when the orbit is bigger. For all experiments the

[5] The computation for potential energy is shown in [240].

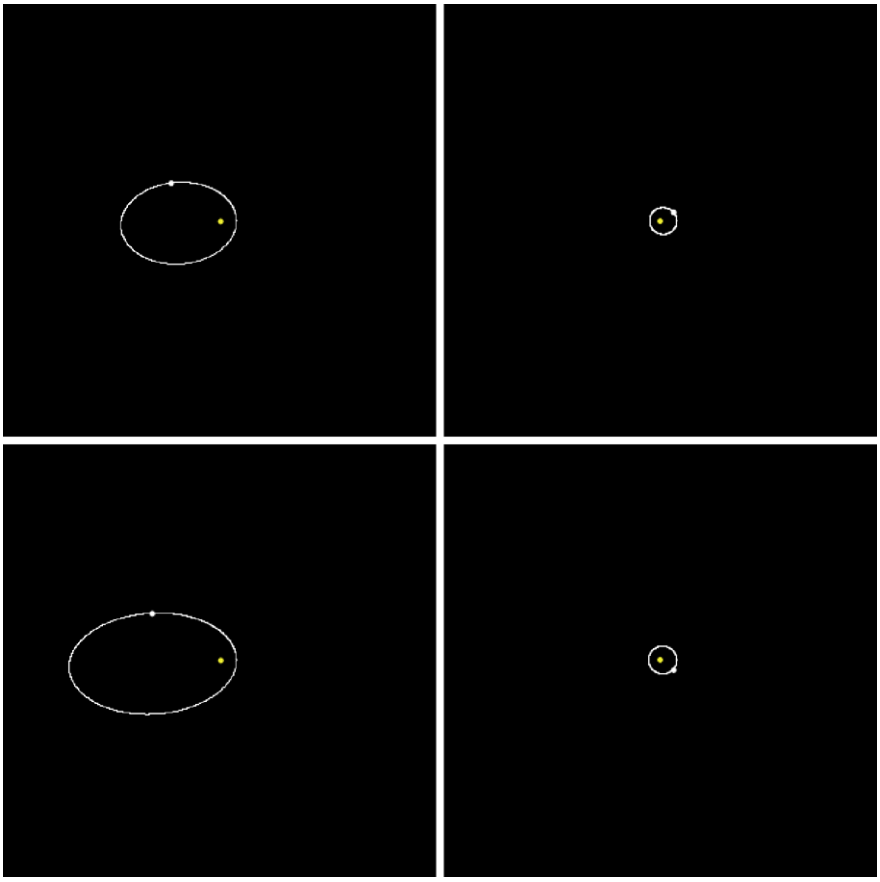

Fig. 2.12: The orbits when `Angular_Motion` is 15 (left) and 10 (right) and the mass of the sun is 100 (top) and 95 (bottom)

linear momenta stay at zero, while the angular momentum and total energy remain very steady. Feel free to try different values of `Angular_Motion`, `Mass_of_Sun` and gravity. Just remember that if the planet moves beyond the perimeter of the environment, the orbit will be very odd!

Reset the mass of the sun to 100, and let's examine the orbits when `Angular_Motion` is five and zero (see Fig. 2.13). These results look unusual. When `Angular_Motion` is five the planet breaks away from the sun. When `Angular_Motion` is zero the results make no sense and the total energy changes radically. What is happening? When `Angular_Motion` is zero there is no angular momentum in the system. Both planet and sun are initialized as stationary objects in space. Hence, due to gravitation, they are attracted to one another. Initially, the planet is 15 units to the right of the sun. Assuming the planet can plunge through the sun without any interac-

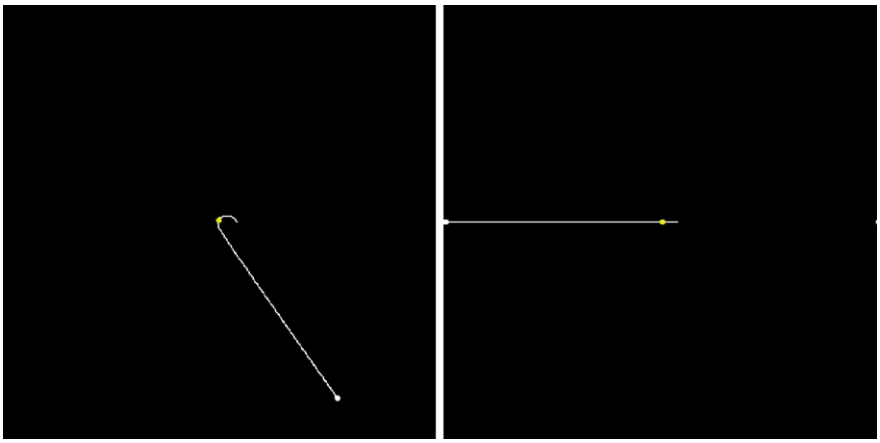

Fig. 2.13: The orbits when `Angular_Motion` is five (left) and zero (right), the mass of the sun is 100, and Δt is 0.1

tion, it should end up being no farther than 15 units to the left of the sun. Yet this is not the case.

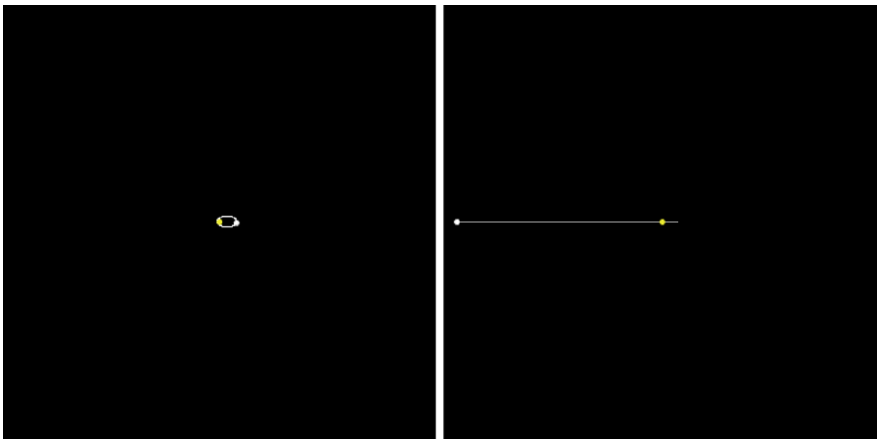

Fig. 2.14: The orbits when `Angular_Motion` is five (left) and zero (right), the mass of the sun is 100, and Δt is 0.001

Could the cause be that Δt is too large? After all, the physics of real solar systems assume continuous time, not discrete time. Reset the mass of the sun back to 100. Set `Time_Step` to 0.001 and try again. Figure 2.14 shows the results. When `Angular_Motion` is five, the orbit now looks reasonable, and both momenta and energy are conserved. However, when `Angular_Motion`

is zero the results are still wrong—the planet continues to move to the left indefinitely. The reason for this behavior lies in the very nature of the gravitational force, which is proportional to $1/r^2$. This means that when the planet is extremely close to the sun, the force becomes huge. This yields a large acceleration and large velocity. In the real world of continuous time, this is not a problem. But with discrete time the particle moves somewhat farther than it should (i.e., the velocity is too large) and the physics is broken. Even when $\Delta t = 0.001$ the simulation is not sufficiently fine-grained.

This also creates a difficulty for implementation on real robots. Again, everything takes time when implemented in hardware, and the faster the hardware the more expensive it becomes. So, this raises an important issue. Can we find a solution that "fixes" the physics in a reasonable manner, without the use of expensive hardware?

One possibility would be to limit the velocity of the robots. This is a natural solution, given that robots have a maximum velocity. In fact, the reason for the planet velocity monitor is to show the velocity that a robot would need to reach. This might not be physically possible. Hence it is natural to wonder if a simple solution of limiting the velocity to that presented by the hardware would solve some of these issues.

Let's reexamine the cases where `Angular_Motion` is 15, the mass of the sun is 100, and the Time_Step is 0.1. The maximum velocity of the planet is roughly 15. Suppose our robot can only move 12 units per time step. Move the Velocity_Maximum slider to 12 and run the simulation.[6] This seems to run quite well, with only a slight change in the linear momentum. However, move the Velocity_Maximum slider to nine and try again. Almost immediately the planet plunges into the sun and remains nearby. The linear momenta oscillate wildly, and the total energy varies a lot. Now try the case where `Angular_Motion` is zero and Velocity_Maximum is set to 12. This is again a disaster, with large changes in linear momenta and total energy. This is also true with Velocity_Maximum set to nine (or one, for an extreme example). Apparently, capping velocity explicitly does not work well in many cases.

There is an alternative solution. Physics works with force laws, and an implicit way to decrease velocity is to cap the force magnitude. Lower forces lead to lower velocities. This is especially important when using a force law that has the form $1/r^2$, since the force magnitude has no limit. Interestingly, no one really knows how the Newtonian force law behaves at small distances, and modifications have been proposed [226]. The slider Force_Maximum allows us to place a maximum on the magnitude of the force. Lower Force_Maximum to one. Because the maximum attractive force is now reduced, we need to use different values of `Angular_Motion` to achieve effects similar to those we obtained before.

The results are shown in Fig. 2.15, with `Angular_Motion` set to nine, four, one and zero. The biggest change in behavior is the precession of

[6] If you are ever unable to achieve a particular value using the slider, open the slider and set the value manually.

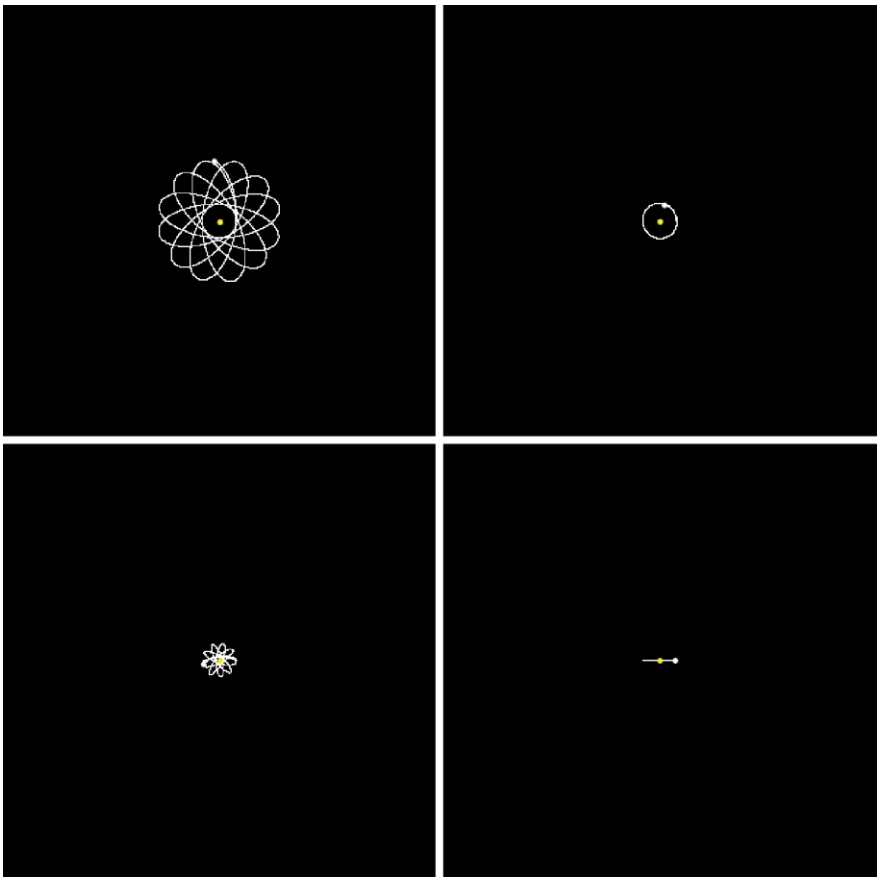

Fig. 2.15: The orbits when Angular_Motion is nine, four, one and zero, while the maximum force is one

the orbits, which did not occur before. Interestingly, this flower-petal shape occurs in real-world orbits, mainly due to perturbations from other planets. Of course, it is occurring in our system because of the imposition of Force_Maximum, which perturbs the elliptical orbit. Force_Maximum has caused a change in behavior for the system, although it is not unreasonable. The planet is still orbiting the sun, and is not breaking away from it.

A nice aspect is that for all these experiments the conservation of energy and momenta hold quite well. The deviations that remain can be reduced further by lowering the Time_Step. The behavior when Angular_Motion is zero is especially nice, with the separation ranging from zero to 15, as we had hoped for originally. The Linear and Angular Momenta graph has values at zero. Also, we have succeeded in our original goal, which was to limit the

velocity. For the four experiments velocity is limited to 5.6, 5, 4 and 9 (when `Angular_Motion` is zero, one, four and nine).

Ultimately, if our robots are fast enough and have very fast sensors, we may not need to impose any force magnitude restrictions in the force laws. However, to further extend the range of behaviors that the robots can perform, using Force_Maximum is a reasonable option, as long as we understand the consequences. In practice, for the problems that we have examined, orbital precession is not an issue, and the use of Force_Maximum has worked quite well. In fact, as we will show in the next chapter, it can be used to our advantage, to help set parameters theoretically.

http://www.springer.com/978-3-642-22803-2

Physicomimetics
Physics-Based Swarm Intelligence
Spears, W.M.; Spears, D.F. (Eds.)
2012, XXX, 646 p. With online files/update., Hardcover
ISBN: 978-3-642-22803-2