Modeling a Bungee Cord Jump: A Classroom Experiment

Introduction: The Challenge

OBJECTIVE: Collect data for a bungee cord (or use data collected by Carroll College students); build, test, and refine a model for a bungee jumper; and estimate the length of bungee cord necessary to achieve a desired length of fall.

In a recent television commercial, a bungee jumper comes so close to the ground that he is able to dip a corn chip he holds in his mouth into a bowl of salsa on the ground without bumping his nose. To do such a jump successfully, the bungee jumper would have to do some serious mathematical modeling, which we proposed to do with students at Carroll College. So, to present the students with "The Bungee Cord Challenge", we acquired some bungee cord material (approximately one-eighth inch in diameter), cut it into various lengths, and gave a piece to each group of students. We also gave them a set of 0.2 kg masses and a metric tape measure. We now present to you the challenge as it was made to students at Carroll College: given a fall distance, how long should you cut the bungee cord so that a 0.2 kg mass, when dropped from that height, would just "kiss" the floor?

Technology Guidelines

NOTE: If you have just finished a module, restart *Mathematica* before executing a new module.
TO OPEN CELLS, put your cursor on the right cell bracket and double click.
INITIALIZATION CELLS
When asked if you want to ". . . automatically evaluate all the initialization cells in the notebook . . . ," respond by pressing the "Yes" button.
TO STOP AN EXECUTION
Select the *Kernel* pull-down menu and click on *Abort Evaluation.*
ORDER OF EXECUTION
Execute cells in the order given. Do not skip any Input cells within a given notebook.
SAVING NOTEBOOKS
You can save anytime to any directory you choose, and it is wise to save often.

However, before you do your final save, delete all your output by selecting the *Delete All Output* selection under the *Kernel* pull-down menu.

EXPERIENCING MAJOR PROBLEMS

Save if appropriate, and then shut down *Mathematica* and start it up again.

Part I: The Experiment

An important element in a complicated model like bungee cord jumping is the spring-like behavior of the bungee cord. In order to characterize this behavior, we asked our students to measure the unstretched length of their piece of cord and then hang 0.2 kg increments of mass from the cord and measure its stretch in meters for each value of hanging mass. So that we could introduce the length of the bungee cord as a variable parameter in the model, the students then calculated the relative stretches for each mass increment, that is, the stretch divided by the unstretched length of the cord. This gives the stretch per unit length of cord for a given mass. Engineers call this relative stretch the strain, and it has units of meters/meter or inches/inch. (Can you see why engineers might want to measure strain instead of stretch? Think about the vertical hanging cables that support the Golden Gate Bridge deck.) We also converted the mass quantities to forces in newtons, multiplying each mass value by 9.8 m/s^2. Since we "normalized" the measure of the stretch by calculating the relative stretches (i.e., the strain), all of the groups got similar results.

The following data set (borrowed from a group of students at Carroll College in Montana) is a list of ordered pairs, in which the first value in each ordered pair is the strain or relative stretch of the bungee cord in meters/meter, and the second value is the corresponding force in newtons.

In[3]:=

```
data = {{0, 0}, {0.0142, 1.96}, {0.0857, 3.92},
  {0.4948, 7.85}};


TableForm[data,
 TableHeadings → {None, {"strain(meters/mete
```

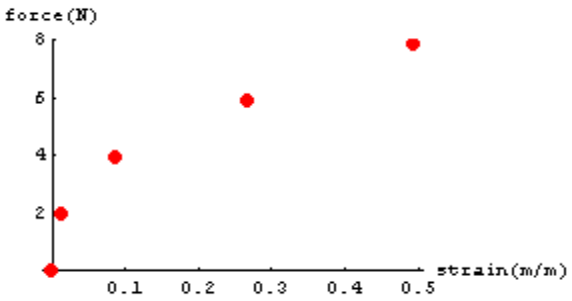
Out[4]//TableForm=

strain(meters/meter)	force(N)
0	0
0.0142`	1.96`
0.0857`	3.92`
0.2662`	5.87`
0.4948`	7.85`

"> About *Mathematica*

In[5]:=

```
p1 = ListPlot[data,
  PlotStyle -> {PointSize[0.04], RGBColor[0.9
  AxesLabel → {"strain(m/m)", "force(N)"}];
```



Part II: Designing a Model

There appears to be a trend that we can capture with a mathematical model, and now we try to find a suitable one using the **Fit[ ]** function. What does the function look like to you? Does it look like a radical or log function? We will try a power function with several different rational exponents, letting **x** represent the strain and **y[x_]** the force corresponding to a strain of **x**.

In[6]:=

```
n = 0.15;

Clear[y];

y[x_] = Fit[data, {x^n}, x]
```

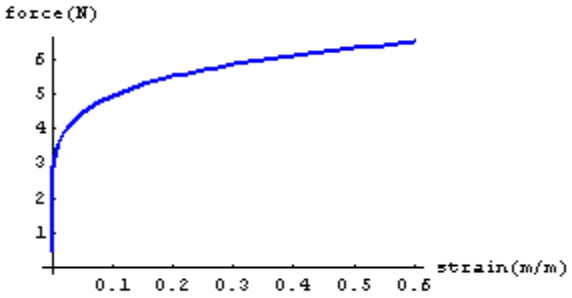
Out[8]=

$$6.97623\, x^{0.15}$$

Next, we plot our model function, save it as **p2**, and then show the plots of the data and the model function together on the same graph.

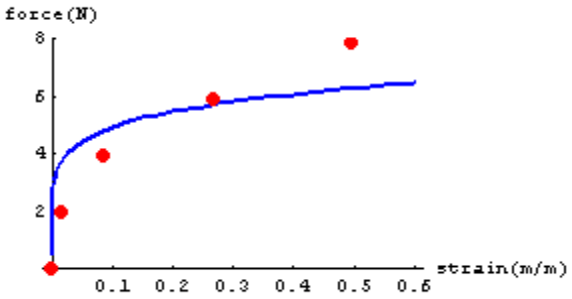
In[9]:=

```
p2 = Plot[y[x], {x, 0, 0.6},
  PlotStyle -> {RGBColor[0, 0, 1], Thickness[0
  AxesLabel → {"strain(m/m)", "force(N)"}];
```



In[10]:=

```
Show[p2, p1];
(* Putting p2 first puts the data points on
  the model graph instead of under it.*)
```



"> About *Mathematica*

Part III: Assessing the Errors

Note: This Part duplicates the discussion of residual errors and the "rms" measure of the error in Part III of the module entitled, "Modeling Change: Springs, Driving Safety, Radioactivity, Trees, Fish, and Mammals." If you are already familiar with the concepts of residual error and "rms" measure of the error, skip directly to "You Try It: Improving the Model."

Our model doesn't appear to be very good! Let's quantify just how good (or bad) it is. You

guessed right; we should calculate the residual errors.

In[11]:=

```
residuals = Table[{data[[i, 1]], data[[i, 2]] -
  {i, 1, Length[data]}];

TableForm[residuals,
 TableHeadings →
  {None, {"Strain (meters/meter)", "Residual
```

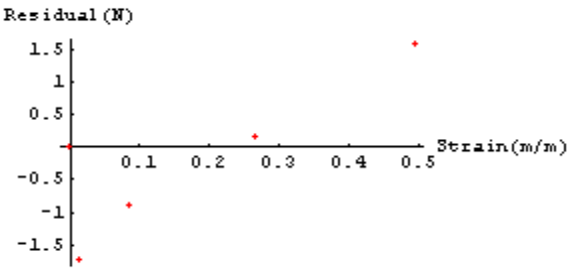
Out[12]//TableForm=

Strain (meters/meter)	Residual Error (N)
0	0
0.0142`	-1.7252275597316387`
0.0857`	-0.9057873646644152`
0.2662`	0.14991784576063605`
0.4948`	1.5725174762103755`

We plot the residuals to see if the errors appear to be random or if there is a discernable pattern.

In[13]:=

```
ListPlot[residuals,
 PlotStyle → {RGBColor[1, 0, 0], PointSize[0.(
 AxesLabel → {"Strain(m/m)", "Residual(N)"}]
```



The plot suggests that there is a pattern to the errors and that the model needs improvement.

To get a global measure of the error for all of the data points in the set, you might be inclined to calculate the average (mean) all of the individual or local residuals, but this can be very misleading.

In[14]:=

```
residavg = Sum[residuals[[i, 2]], {i, 1, Lengt
  Length[data]
```

Out[14]=

```
-0.181716
```

As you can see, the mean residual error is small relative to the average size of the forces in the data set. On this basis, we might be led to believe that our model isn't so bad after all. Wrong! The problem is that the individual or local errors are actually sizable when compared to the measured forces, but because some of them are positive and some are negative, they tend to cancel each other when we add them together to calculate their mean value. Look at the values in the list of residual errors, or look at the vertical differences between the data points and the fit function on the graph, and you can easily see that the average of the residual errors is misleading.

There is an infinite number of ways to address this canceling problem, but one of the most common is to calculate the sum of the squares of the residuals and try to find the smallest or least value of this sum of the squares of residuals. Hence, we have the term "least squares," which you may have heard before. Squaring the residuals removes the canceling effect that occurs when we add them together for a measure of the global error. The *Mathematica* **Fit[ ]** function uses least squares, and some variations of it, to find a best-fit function for a set of data. Finding the minimum value for the sum of squared residuals is a problem that can be solved using calculus, and you will study this problem later on, but for now you can do it by trial and error. Let's get back to our bungee cord challenge. What we want to do now is calculate the sum of the squares of the residuals for our data set.

In[15]:=

```
residsquaresum = Sum[residuals[[i, 2]]^2, {i,
```

Out[15]=

```
6.29215
```

Before we compare this value, we calculate the mean of the squared residuals.

In[16]:=

```
msresiduals = residsquaresum / Length[data]
```

Out[16]=

```
1.25843
```

Comparing the average of the squared residuals with the average of the force values in the data set would be like comparing apples with oranges because **msresiduals** is an average of squares, whereas the force average is not. To make a fair comparison, we take the square root of the mean of the squares.

In[17]:=

```
rmsresiduals = Sqrt[msresiduals]
```

Out[17]=

```
1.1218
```

The value that we calculate in the preceding step is the root of the mean of the squares of the residuals, oftentimes called the root-mean-square or "rms" value of the residuals.

You Try It: Improving the Model

Now we have a fair number to compare with the average of the measured force values. As we initially expected, this comparison demonstrates that our model needs improvement. We will leave that up to you, but, to help out, we group all of the commands that you need for the error analysis into the cell that follows. Use trial and error to find a better model function by changing the value of the exponent *n* (highlighted in red) to minimize the sum of the squared residuals, which will also minimize the rms measure of the error.

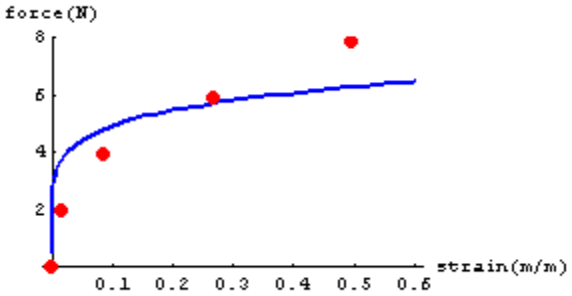
(Before you proceed, we offer some comments about errors in modeling: in mathematical modeling, you cannot completely eliminate errors. Random errors occur in measurements due to the limited precision of all measuring instruments, and systematic errors, that is, errors that follow a pattern, occur due to shortcomings of the model and/or possible defects in the measuring tools. Systematic errors can often be reduced or eliminated by refining the model and by repairing and/or calibrating the measuring equipment, but random errors are unavoidable. We can reduce the magnitudes of random errors by using more precise instruments, but we cannot eliminate them; the errors may be smaller, but they are always present. With this in mind, select a value for *n* that minimizes **residsquaresum**, the sum of the squares of the residuals.)

```
"> About Mathematica
```

In[18]:=

```
n = 0.15;
Clear[y];
y[x_] = Fit[data, {x^n}, x];
 Print["The model is ", y[x]];
p2 = Plot[y[x], {x, 0, 0.6},
   PlotStyle -> {RGBColor[0, 0, 1], Thickness[
   AxesLabel → {"strain(m/m)", "force(N)"},
   DisplayFunction → Identity];
Show[p2, p1, DisplayFunction → $DisplayFuncti
residuals =
  Table[{data[[i, 1]], data[[i, 2]] - y[data[[
   {i, 1, Length[data]}];
residsquaresum = Sum[residuals[[i, 2]]^2,
   {i, 1, Length[data]}];
 Print["The sum of the squares of the resid
  residsquaresum];
```

The model is $6.97623\,x^{0.15}$



The sum of the squares of the residuals is
6.29215

"> About *Mathematica*

Part IV: You Can't Push a Rope

We need to address one final item before we take the bungee cord challenge. You may have heard the expression, "You can't push a rope." In the context of our bungee cord, this means that if the stretch is negative (i.e., if we try to push on the ends of the bungee) the bungee

simply folds, and the force in it is 0. To accommodate this, we modify our model to say that if the stretch is negative, the force is zero. Here's how we do it in *Mathematica*.

In[27]:=

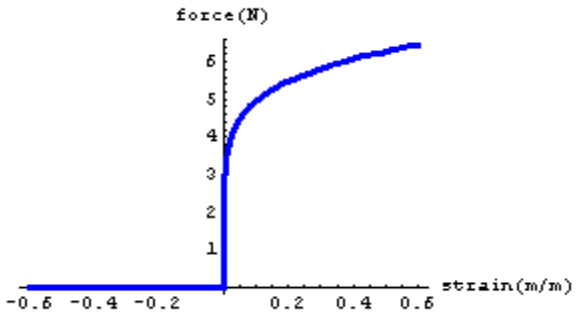
```
Clear[bungeeforce];

bungeeforce[x_] = Which[x < 0, 0, x ≥ 0, y[x]];
```

Observe the graph of the force in the bungee cord for negative as well as positive values of x, the strain in the bungee cord.

In[29]:=

```
Plot[bungeeforce[x], {x, -0.6, 0.6},
 PlotStyle -> {RGBColor[0, 0, 1], Thickness[0.
 AxesLabel → {"strain(m/m)", "force(N)"}];
```



"> About *Mathematica*

This model provides a nice, real-world example of a piecewise-defined function. The extended piecewise-defined model is useful for solving the differential equation of motion to obtain a more detailed description of the motion of the bungee jumper during the fall.

Part V: Using the Model

Chapter 6, Section 6, Exercise 25 Extension

We're ready to tackle the bungee cord challenge. We will use the Work-Energy Theorem, a principle from physics, to calculate the fall distance for a given length of bungee cord. The Work-Energy Theorem states that when an object (in this case the 0.2 kg mass) moves

through a distance (the fall), the total work done on the object during the motion is equal to the difference in the kinetic energy of the mass between the beginning and end of the fall. If W_{total} is the total work done and K is the kinetic energy, then the Work-Energy Theorem says that from the beginning to the end of the fall, $W_{total} = \Delta K$. Exercise 25 in Section 4 of Chapter 5 (in Section 5 of Chapter 5 in Early Transcendentals) asks you to derive this result using Newton's Second Law of Motion and the Chain Rule.

The kinetic energy of a mass is the energy that it has by virtue of its motion, and it is calculated by $K = \frac{1}{2} m v^2$, where m is the mass of the object and v is its speed. In our problem, the mass is dropped from rest and, at the bottom of the fall, it comes to rest again but just for an instant before the stretched bungee begins to pull it back up. Therefore, v=0 at the beginning and at the end of the fall, so the change in kinetic energy is 0, that is, ΔK=0. Consequently, the total work done on the mass during the fall must be 0, i.e., W_{total} =0.

The forces that do work during the fall are the weight of the mass, the pull of the bungee cord, and friction forces due to air resistance and the rubbing together of the materials that make up the bungee cord. We will ignore the friction forces and consider only the work done by the weight force and the bungee force.

The work done by gravity is easy to calculate because the weight force is constant during the fall; therefore,

$W_{gravity}$ =(mg)d=(0.2 kg)(9.8 m/sec^2)d=1.96d (in Joules),

where d is the fall distance. Now we do it with *Mathematica*.

In[30]:=

```
workgravity = 0.2 * 9.8 * d
```

Out[30]=

```
1.96 d
```

The work done by the bungee cord is negative because, during the fall, the force is in the direction opposite the motion; that is, the bungee cord pulls up on the mass as it falls downward. We calculate the work as follows: $W_{bungee} = -\int_0^d \text{bungeeforce}\left(\frac{s-L}{L}\right) ds$, where s is the distance fallen, L is the unstretched length of the bungee cord, and $\frac{s-L}{L}$ is the strain in the bungee (i.e., the stretch divided by the unstretched length).

The bungee cord doesn't stretch until the distance fallen is greater than the unstretched length of the bungee cord, that is, until s>L. Since the bungee force is defined piecewise, we need to

divide the work integral into two parts. In the first part, when s<L, the bungee cord force is 0 and so the work done is 0. In the second part, when s>L, the bungee cord force is given by our **y[x_]** function. We now use *Mathematica* to calculate this second integral.

In[31]:=

```
Clear[workbungee];

workbungee = -∫_L^d y[(s - L)/L] ds
```

Out[32]=

$$-\frac{1}{d-1.\,L}\left(6.97623\left(0.+0.869565\left(-1.+\frac{d}{L}\right)^{1.15}\right)(d-L)\,L\right)$$

We're almost done. Let's calculate the total work, set it equal to 0, and solve for *d* in terms of *L*.

In[33]:=

```
Clear[totalwork, d];

totalwork = workgravity + workbungee
```

Out[34]=

$$1.96\,d-\frac{1}{d-1.\,L}\left(6.97623\left(0.+0.869565\left(-1.+\frac{d}{L}\right)^{1.15}\right)(d-L)\,L\right)$$

If you try to set **totalwork** equal to 0 and use *Mathematica*'s **Solve[ ]** command to find L in terms of *d*, our original challenge, you will find that it won't work. Rats! But we're too far along to give up now. Let's try this: assign different values to *d*, use the **FindRoot[ ]** command to calculate *L* for each of the *d* values, then plot L versus *d*, and see if there is a pattern. There should be.

First, let's see how the **FindRoot[ ]** command works. We assign a value to *d* and then use **FindRoot[ ]** to determine the corresponding value for *L*.

In[35]:=

```
d = 1;
```

```
FindRoot[totalwork == 0, {L, 0.1}]
```

Out[36]=

```
{L → 0.646321}
```

"> About *Mathematica*

The answer seems a reasonable one. The model says that if the fall distance is 1 meter, then we should cut the bungee cord so that the part of the cord that stretches during the fall is 0.655 meters long. Don't forget that you need a length of cord in addition to the calculated length so that you can tie the cord to the mass and to the support.

Note also that to determine *d*, the fall distance, you have to be sure to take the distance from the support to the floor and subtract the length of the mass. We want *d* to be the stretched length of the cord when the mass just kisses the ground; therefore, the total distance from the support to the floor will be *d* plus the length of the mass.

Part VI: Making Predictions

In this Part, we build a table of ordered pairs in which the first entry in each ordered pair is the fall distance *d*, and the second element is the required length of cord (not including the tie ends). The output from **FindRoot[ ]** is a list with the rule L→0.655219 inside. To extract the number 0.655219, we use a pair of double square brackets as follows.

In[37]:=

```
soln = FindRoot[totalwork == 0, {L, 0.1}];

soln[[1, 2]]
```

Out[38]=

```
0.646321
```

"> About *Mathematica*

Now, we use the **Table[ ]** command to build a set of ordered pairs, in which the first element of each ordered pair is the fall distance *d*, and the second element is the required bungee cord length, not including the tie ends.

In[39]:=

```
Clear[d];

dversusL = Table[{d, FindRoot[totalwork == 0, {
  {d, 0.5, 10, 0.5}];

TableForm[dversusL,
 TableHeadings →
  {None, {"Fall Distance (m)", "Bungee Lengt
```

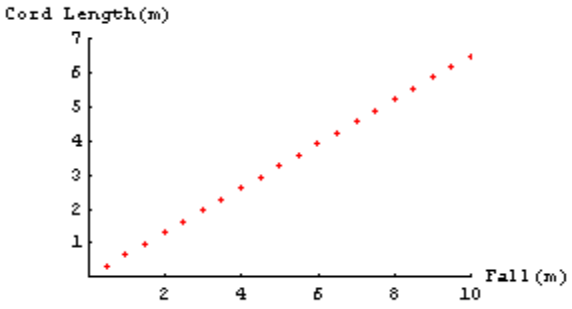
Out[41]//TableForm=

Fall Distance (m)	Bungee Length (m)
0.5`	0.32316068903133177`
1.`	0.6463213780626637`
1.5`	0.9694820670939952`
2.`	1.2926427561253273`
2.5`	1.6158034451566587`
3.`	1.9389641341879906`
3.5`	2.2621248232193225`
4.`	2.585285512250653`
4.5`	2.9084462012819863`
5.`	3.2316068903133175`
5.5`	3.5547675793446496`
6.`	3.877928268375981`
6.5`	4.201088957407313`
7.`	4.524249646438645`
7.5`	4.847410335469977`
8.`	5.170571024501308`
8.5`	5.4937317135326404`
9.`	5.816892402563973`
9.5`	6.140053091595304`
10.`	6.463213780626637`

Let's plot them.

In[42]:=

```
p3 = ListPlot[dversusL,
  PlotStyle -> {PointSize[0.02], RGBColor[0.9
  AxesLabel → {"Fall(m)", "Cord Length(m)"},
  AxesOrigin → {0, 0}, PlotRange → {{0, 10}, {0
```



It looks like a straight line through the origin, which is rather surprising. You know what to do next.

In[43]:=

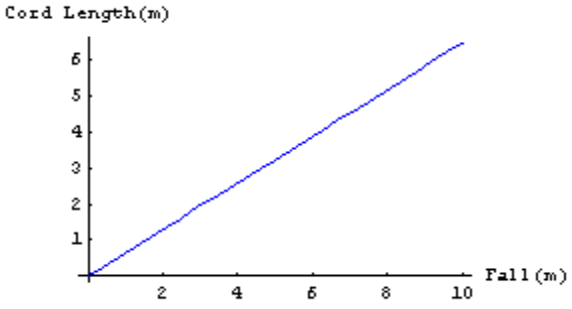
```
L[d_] = Fit[dversusL, {d}, d]
```

Out[43]=

```
0.646321 d
```

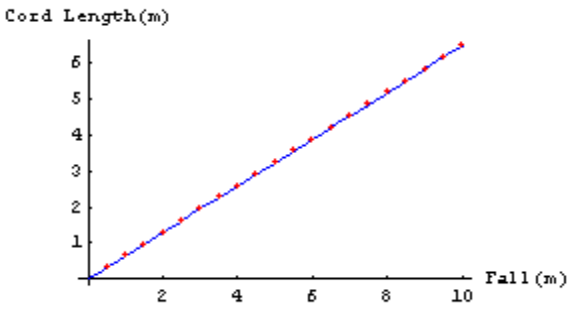
In[44]:=

```
p4 = Plot[L[d], {d, 0, 10},
  PlotStyle -> {RGBColor[0, 0, 1], Thickness[0
  AxesLabel → {"Fall(m)", "Cord Length(m)"}]
```



In[45]:=

```
Show[p4, p3];
```



Part VII: Testing the Results

The Carroll College students who tested the model found that it worked extremely well. For various fall distances, the cord was cut to just the right length so that the mass kissed the floor when dropped. This provided a good validation of the model.

You Try It: Do Your Own Experiment

Go to a sporting goods store and get some small bungee cord material (one-eighth inch in diameter or so) and a small inexpensive metric tape measure. (In sporting goods, bungee cord is sometimes called shock cord.) Then, go to your physics department and borrow some calibrated masses. See if you can replicate this modeling exercise. The data you get will likely differ from ours because your bungee cord material probably won't be the same. But, we hope your results will be as good.

About *Mathematica*

The semicolon after a *Mathematica* command suppresses the output. In the preceding command cell, we grouped the data list command and the **TableForm[ ]** command together in one cell with a semicolon after the definition of the data list and no semicolon after **TableForm[ ]**. In this way the data table is displayed for easy reading, and the data list is suppressed. If you remove the semicolon, both forms of the data are displayed. Go Back.

You can include comments in a *Mathematica* command cell by enclosing the comment inside (* *), as we did in the preceding command. This can be useful for documenting your work, which is good programming practice. When *Mathematica* executes a command, it ignores everything enclosed by the (* *) markers. Go Back.

Among other things, *Mathematica* is a programming language. When we group a series of commands together in a single cell, as in the cell that follows, we are essentially writing a *Mathematica* program. Although we have not done it here, we could give this group of commands a symbol name and use it like any other *Mathematica* command. We do this with the **Block[ ]** command or the **Module[ ]** command. To learn more about programming in *Mathematica*, see "Overview of *Mathematica*: Making Your Own Commands," included in this supplement. Go back.

You may have noticed that when a semicolon follows a graphics command like **Plot[ ]** or **ListPlot[ ]**, the graphics output is not suppressed as it is for most other *Mathematica* commands. If you want to suppress a graphics display, do what we did in the preceding set of commands. As an option in the **Plot[ ]** command, we type **DisplayFunction→Identity**, and the display is suppressed. Later, when we want to display the graphics in the **Show[ ]** command, we type **DisplayFunction→$DisplayFunction** as an option. Specifying options in a graphics command is very useful for tailoring your graphics output. To learn more about the various options for graphics commands, go to the Help Browser, type **Plot**, and then scroll down to see some of the options that are available. Go Back.

The **Which[ ]** command is very useful for functions like the bungee cord force that are piecewise defined. Many mathematical operations can be done in *Mathematica* on functions that are defined in this way. For more information about the **Which[ ]** command and related commands like **If[ ]** and **Switch[ ]**, go to the Help Browser, select Programming in the left column of the dialog box, and then select Flow Control in the center column. Go Back.

The **Solve[ ]** command is used to solve algebraic equations. When the equations are non-algebraic, however, the **Solve[ ]** command often won't work. This is the case with the problem we are trying to solve. The next alternative is to use a numerical approach with the **FindRoot [ ]** command. This command almost always finds a numeric estimate of a root, but it won't work on equations that have variables in them, other than the unknown. The first argument in **FindRoot[ ]** is the equation to be solved. (Note that mathematical equalities are expressed with the double equals, "==", because the single equal sign is reserved for the assignment of values or expressions to symbol names. An example is the assignment to **totalwork** in one of the preceding cells.) The second argument of **FindRoot[ ]** is a list; the first element of the list is the variable we wish to solve for, and the second is an initial estimate of the solution. We selected an initial estimate of 0.1 in our use of **FindRoot[ ]** in the preceding cell. To learn more about **Solve[ ]** and **FindRoot[ ]**, go to the Help Browser and type **Solve** and/or **FindRoot**. Go Back.

The output list that we get when we use **FindRoot[ ]** in the preceding cell has one element with two parts, the variable name and the solution value for that variable. Therefore, to extract the solution, we use pairs of double brackets with part designations, as in the preceding cell.

To learn more about extracting elements from expressions and lists, see "Overview of *Mathematica*: Parts of Lists," and go to the Help Browser and type **List** and/or **Part**. Go Back.