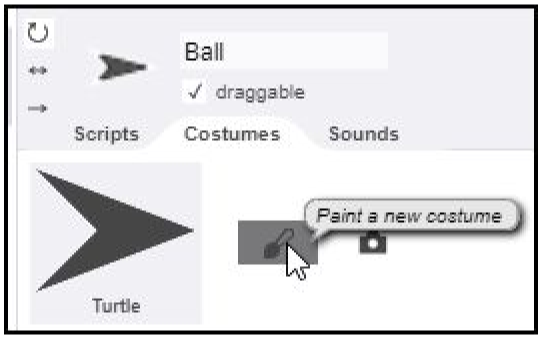
Module 2. Periodic Motion

The concepts of *frequency* and *period* were introduced in the previous module. Events in which the period of each occurrence of the event is the same as the one before are said to be *periodic*. In the previous module, the period of each rotation of the turtle was the same as the one before.

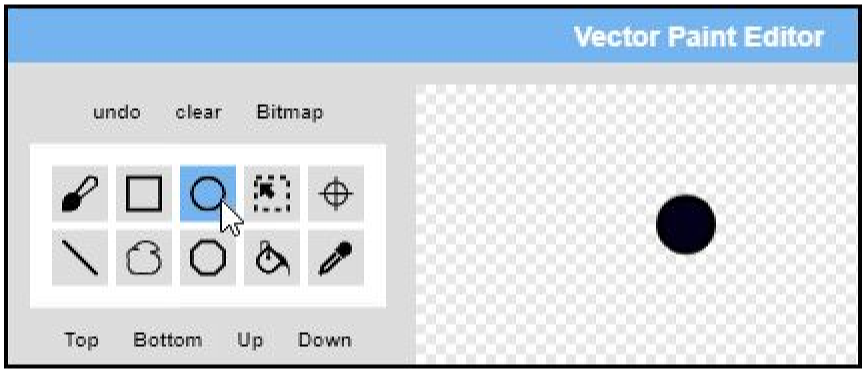
The motion of naturally vibrating objects is often periodic. For example, the up-and-down motion of a ball on a spring is periodic. The same up-and-down motion repeats over and over again as the ball bobs up and down. A computer can be used to simulate this motion.

Topic 2.1 Creating a Ball Sprite

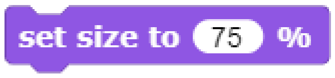
The turtle is the default costume for sprites. Other costumes can be created in the *Paint Editor*. This option is accessed under the *Costumes* tab. Since the sprite will be used to emulate a ball, enter the name "Ball" in place of the default name of "Sprite". Then click the paint brush icon to enter the *Paint Editor*.



Then draw a ball in the Paint Editor. Once a solid ball is drawn, click OK to confirm and exit the Paint Editor.

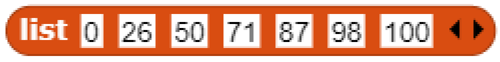


Exploration 2.1 Experiment with the **Set Size** code block, found under the violet *Looks* palette, to adjust the size of the ball.



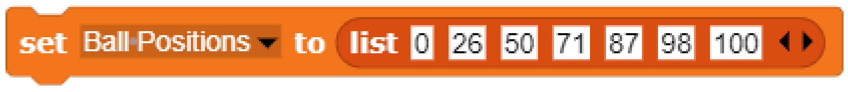
Topic 2.2 Positioning the Ball

In order to simulate the motion of a ball attached to a spring, a list of the vertical positions of the ball at various points in time is needed. To simplify matters, the ball is assumed to move from 0 on the vertical axis to an upward limit of 100 when the spring is fully compressed. The following ball positions corresponding to the upward movement of the ball are entered into a list.



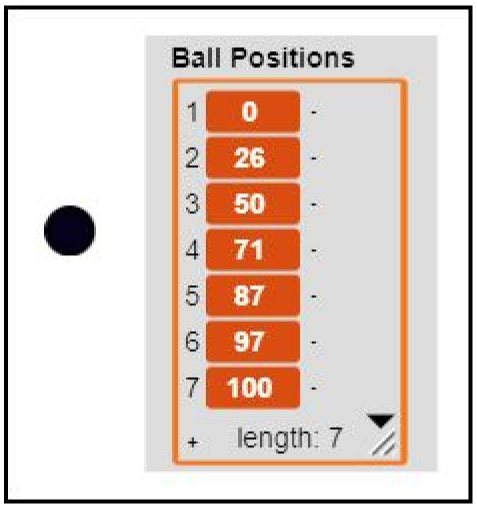
```
list 0 26 50 71 87 98 100
```

Then a variable named *Ball Positions* is created. The value of this variable is set to the previously created list of ball positions.



```
set Ball Positions to list 0 26 50 71 87 98 100
```

Once this is done, a table of the ball positions will appear on the stage.

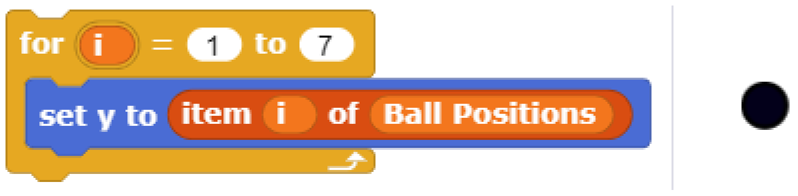


Ball Positions

1	0
2	26
3	50
4	71
5	87
6	97
7	100

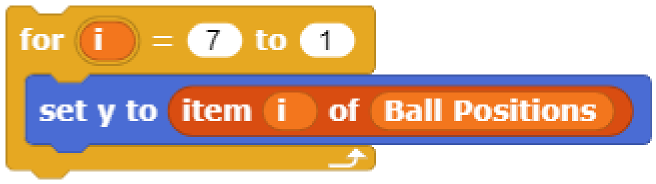
length: 7

The following loop sets the ball to the vertical positions corresponding to each item in the *Ball Position* table. During the first iteration of the loop, the ball is set a vertical position of 0. In the second iteration of the loop, the ball is set to a vertical position of 26. The process continues until the ball is set to a vertical position of 100 during the seventh and final iteration of the loop.



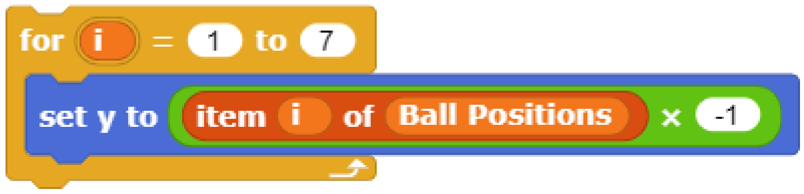
```
for i = 1 to 7
    set y to item i of Ball Positions
```

The ball is now at a vertical position of 100, when the simulated spring is at its maximum compression. The ball now begins to fall downward. The downward movement can be simulated through a loop that begins at Item 7 of the Ball Position table, and works its way back through the table until the ball is once again at a vertical position of 0 when Item 1 in the table is reached.



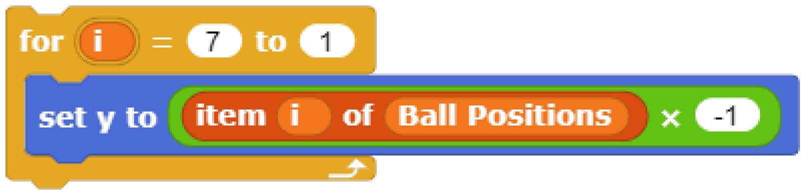
```
for i = 7 to 1
    set y to item i of Ball Positions
```

The ball presumably will continue to fall as the spring stretches. This continued downward movement can be simulated by walking through the Ball Position table once again. However, this loop multiples each of the ball positions by minus one (- 1). This will cause the ball to move to -26 during the second iteration of the loop, continuing to the seventh and final iteration resting at – 100.



```
for i = 1 to 7
    set y to item i of Ball Positions × -1
```

The fourth and final loop reverses the process to move the ball upward until it rests at 0 once again.



```
for i = 7 to 1
    set y to item i of Ball Positions × -1
```

The complete process in which the ball moves through one complete upward and downward movement until it rests at 0 once again can be described as one *cycle* of movement.

Exploration 2.2 Combine all four loops to move the ball through a complete cycle of upward and downward movement. Place the combined script in a **Forever** loop to create a continuous up and down movement.

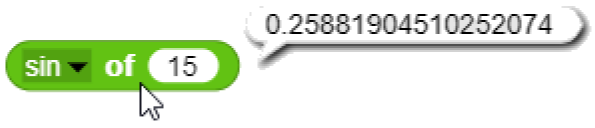
Topic 2.3 Controlling Movement with a Mathematical Function

The values chosen for the table of Ball Positions assume that as the ball moves upward, it will move a smaller and smaller distance during each moment of time as the spring compresses. When the ball begins moving downward again, it moves a greater and greater distance during each moment in time as it accelerates toward the midpoint. As the ball stretches the spring to its farthest point as it moves downward, it moves shorter and shorter distances as the spring is extended.

Thus, the ball moves shorter and shorter distances as it approaches the upward and downward endpoints and moves greater distances as it moves toward the midpoint. The behavior of the ball in this simulation is a good approximation of the way in which a ball on a spring actually behaves. In fact, any naturally vibrating object tends to have a similar behavior.

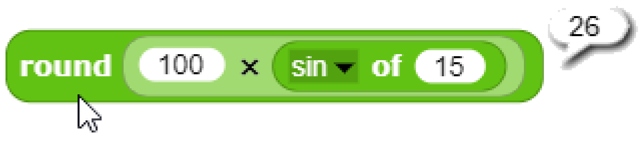
For example, a swing tends to slow as it reaches the top arc of its upward swing and accelerates as it moves through the midpoint. Similarly, the tine of a tuning fork exhibits similar behavior as it moves back and forth. An object must move back and forth at least twenty times per second or more before the vibration is perceived as sound. The motion of objects that are vibrating more slowly, however, can be more easily seen as they move back and forth, making it easier to understand the patterns of motion that are common to all naturally vibrating objects.

The positions of a vibrating object can be described by a mathematical function known as a *sine function* (abbreviated *sin*). The sine function in Snap*!* can be located under the green *Operators* palette. The value returned by the *sin of 15 degrees* is "0.258".



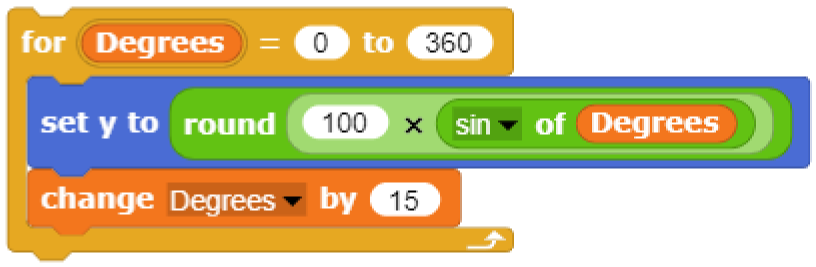
```
sin of 15
0.25881904510252074
```

When this value is multiplied by 100 and rounded, the result is "26", the same value that is the second item in the table of Ball positions.



```
round (100 × (sin of 15))
26
```

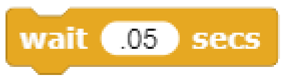
The value obtained for 30 degrees is 50, and the value obtained for 45 degrees is 71. These are the values that are the third and fourth items in the table of Ball Positions. In fact, the numbers in the table of Ball Positions correspond to a table of sine functions, multiplied by 100 and rounded.



```
for Degrees = 0 to 360
    set y to round (100 × (sin of Degrees))
    change Degrees by 15
```

Because this is the case, the sine function can be used to control the position of the ball as it moves upward and downward. The sine function is applied to degrees ranging from 0 through 360 degrees, changing in 15-degree increments during each iteration of the loop.

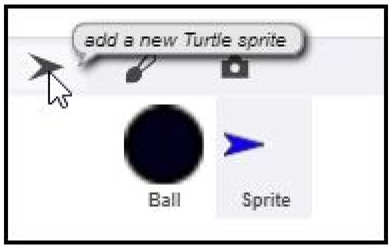
Exploration 2.3 Controlling the Movement of the Ball with a Mathematical Function. Insert a **Wait** code block into the loop to slow the movement of the ball and observe the individual movements of the ball during each iteration of the loop. Experiment with values ranging from one second to a twentieth of a second (0.05 seconds).



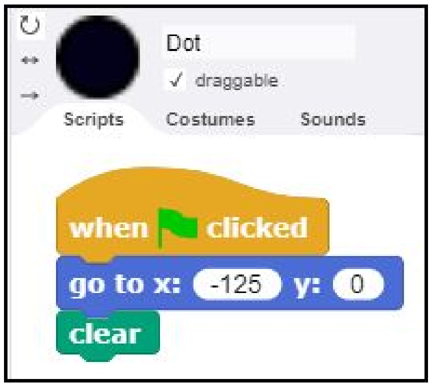
```
wait .05 secs
```

Topic 2.4 Plotting the Movement of the Ball

A graph of the movement of the ball is helpful in understanding the nature of this movement. Until now, only a single sprite has been employed. A second sprite can be added to plot the points that correspond to the movement of the ball. The sprite icon beneath the stage is used to add a new sprite.



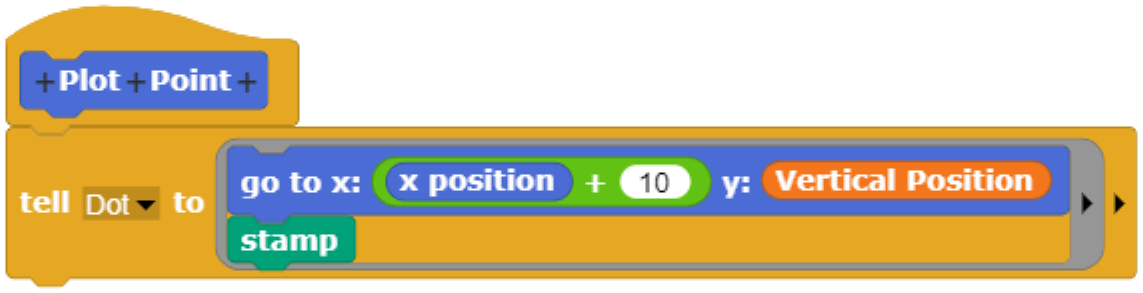
Each sprite can have its own scripts and its own costumes. The second sprite will be used to place a dot to mark the vertical location of each movement of the ball. Therefore, it is given a solid circle smaller than the ball as a costume and assigned the name, “Dot”. A script is created for the Dot sprite that positions it on the left side of the stage and clears the screen.



```
when [flag] clicked
go to x: -125 y: 0
clear
```

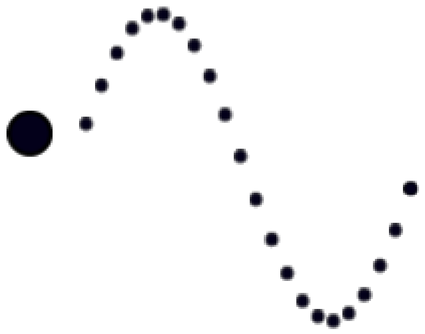
The script for the *Ball* sprite can now be modified to plot a point to mark each vertical position of the ball. A custom code block, **Plot Point**, is created to perform the plotting function. A new variable, *Vertical Position*, provides a means of communicating the position of the ball with the **Plot Point** code block.

The **Plot Point** code block has two commands. It tells the Dot sprite to go to the current vertical position of the Ball sprite and then stamp a copy of its costume at that location. That creates a dot to mark the location of each ball position. The Dot sprite also moves over horizontally ten steps each time the **Plot Point** procedure is run so that the dots are spaced out across the stage.



```
+Plot+Point+
tell Dot to
    go to x: (x position + 10) y: Vertical Position
    stamp
```

When the procedure is run, several observations can be made. First, the plot of the points marking the location of the ball make it apparent that the ball is traveling a greater distance as it moves through the midpoint, while the distance traveled as it approaches the top and bottom end points is less. This is to be expected, since the graph reflects the values in the table of sine numbers. However, the graph highlights this effect visually.

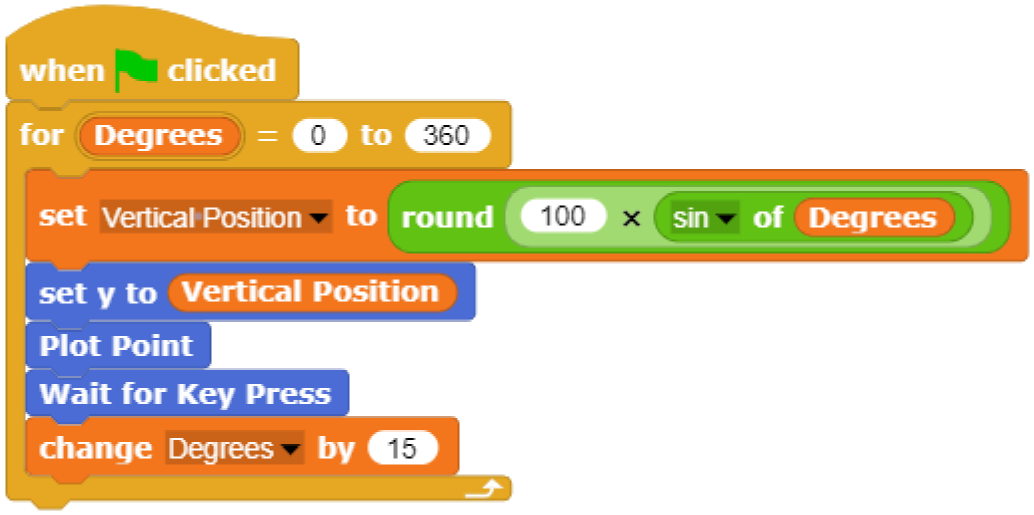


Second, the plotted points create the curve known as a *sine wave*. Finally, the plot of one complete cycle has one positive peak and one negative peak. When several cycles of a sine wave are plotted, the number of cycles can be determined by counting the high peaks, since each cycle has one peak.

Exploration 2.4 Plotting the Movement of the Ball. Explore the effect of adjusting the spacing of the plotted points. How does this affect the visual appearance of the graph produced?

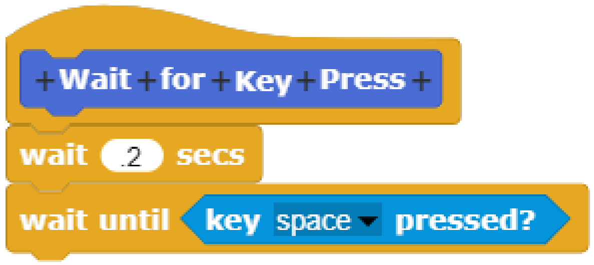
Topic 2.5 Stepping through Plotted Points

The correspondence between the location of the ball and the location of the plotted point is easier to observe when there is a pause after each point is plotted. This pause can be achieved by inserting a custom **Wait for Keypress** code block into the loop.



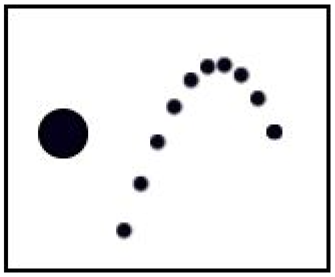
```
when [flag] clicked
for (Degrees) = 0 to 360
    set Vertical Position to round (100 × sin of (Degrees))
    set y to (Vertical Position)
    Plot Point
    Wait for Key Press
    change Degrees by 15
```

The custom **Wait for Keypress** code block consists of two commands. The **Wait** code block in the procedure waits until the spacebar is pressed before continuing.



```
+Wait+for+Key+Press+
wait .2 secs
wait until <key space pressed?>
```

Depending on how long the spacebar is depressed, in some cases a second or third point may be plotted before the spacebar is released. To prevent that occurrence, a fifth of a second (0.2 second) delay is inserted. This delay is usually sufficient to prevent multiple points from being plotted while the spacebar is depressed, but if necessary, the length of delay can be increased.



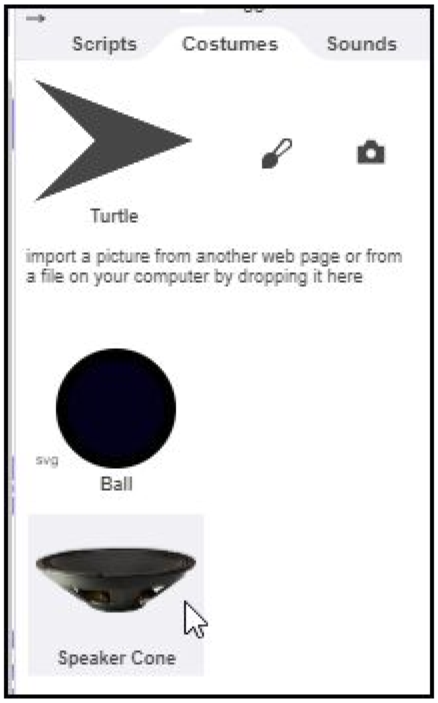
Stepping through the plot one point at a time makes it possible to observe the way in which each point plotted corresponds to the vertical position of the ball at that moment in time.

Exploration 2.5 Stepping through Plotted Points. In addition to stepping through plotted points one point at a time, explore the effect of replacing the **Wait for Keypress** code block with a Wait statement.

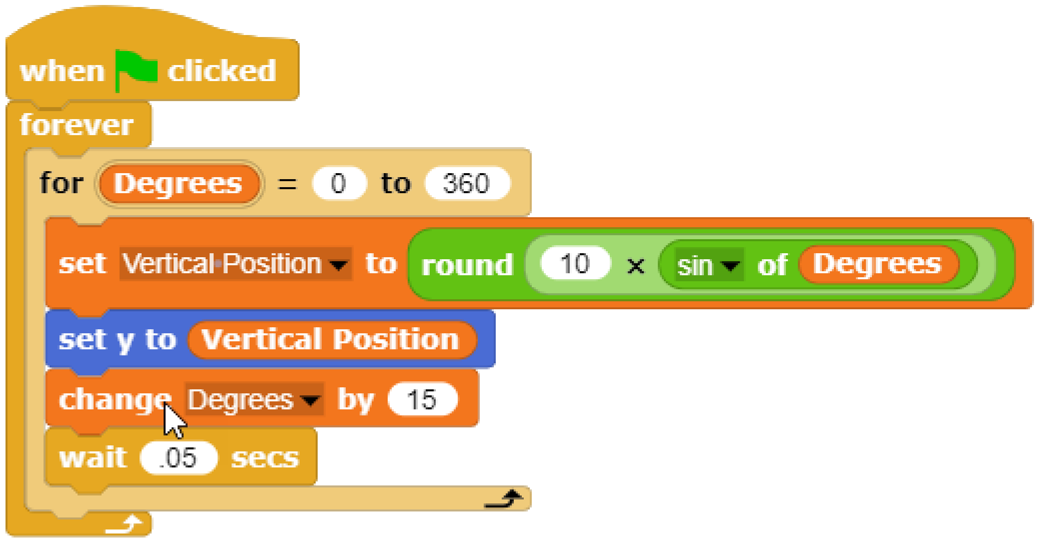
Topic 2.6 Simulating the Movement of a Speaker

Computers use the sine function to synthesize tones. The signal is transmitted to the computer speaker. The varying electrical signal controls an electromagnet that moves the diaphragm of the cone back and forth.

This movement can be simulated by replacing the image of a ball with a speaker cone. An external image of a speaker is imported into Snap*!* where it can be used as a costume for the sprite in place of the image of the ball.


Scripts
Costumes
Sounds
Turtle
import a picture from another web page or from a file on your computer by dropping it here
svg
Ball
Speaker Cone

Two adjustments to the procedure controlling the simulation make it more realistic. The distance that an actual speaker cone travels is not great. The distance that the cone in the simulation travels can be reduced by changing the figure used to multiply *Sin of Degrees* from “100” to “10”.



```
when [flag] clicked
forever
  for Degrees = 0 to 360
    set Vertical Position to round (10 × sin of Degrees)
    set y to Vertical Position
    change Degrees by 15
    wait .05 secs
```

Insertion of a short delay of about a twentieth of a second (0.05 seconds) at the end of the loop makes it easier to follow the movement of the speaker as it travels up and down.

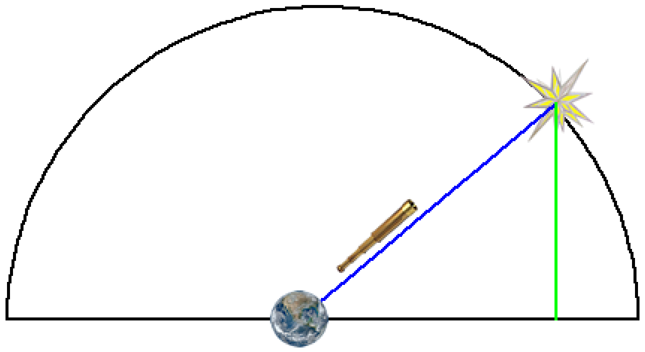
In the simulation, the farthest upward movement of the speaker cone corresponds to the highest peak of the sine wave. Similarly, the farthest downward movement of the speaker cone corresponds to the farthest downward movement of the speaker cone. In other words, the physical position of the speaker directly corresponds to the peaks and valleys of the waveform. This observation is crucial to an understanding of sound and vibration.

Exploration 2.6 Simulating the Movement of a Speaker. Explore the effect of changing the distance that the image of the speaker cone in the simulation travels. Try substituting other values for the delay in the **Wait** code block to explore the effect of this change. Create a procedure that measures the time that it takes to complete one up-and-down cycle.

Topic 2.7 Origins of the Sine Function

The sine function described in preceding sections were first discovered by Greek astronomers. Early astronomers created a map of the stars in the sphere surrounding the earth. Charting the skies required geometry that mapped points on a circle.

The process involved calculation of the angles and sides of right triangles. In the illustration below, the ratio of the green line divided by the blue line yields the value now known as the sine of the angle opposite the green line.



The word *sine* comes from **jīvā**, the Sanskrit for chord (a line that connects two points on a circle). This word was mistakenly transcribed as “jiba” by Arabic astronomers. The word *jiba* means *fold*. The Arabic transcription, in turn, was translated into Latin as *sinus* (i.e., a cavity).

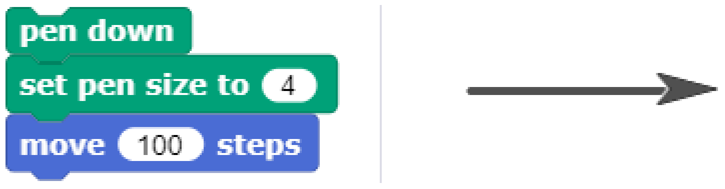
In this way, the work of stargazing astronomers led to discovery of relationships that also have proven to be useful in describing the motion of vibrating objects. Digital computers with capabilities beyond the most remote dreams of ancient astronomers now use these functions to synthesize artificially generated tones.

Topic 2.7 Reconstructing the Sine Function

In the preceding topics, the values for sine have been provided, either in the form of a table or in the form of the built-in **Sin** code block provided in Snap*!* To provide a better understanding of the origins of the sine function that underlies synthesized tones, the sine function will be reconstructed in this section.

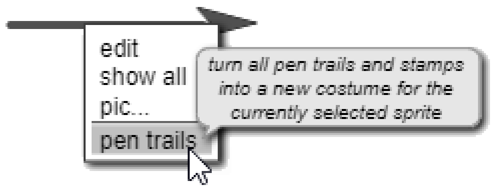
The sine function can be calculated by observing the vertical position of a point on the edge of a circle. Creation of this simulation begins with construction of a line that will be used to sweep out a circle as it rotates.

To simplify the mathematical calculations, a line that is 100 steps long will be drawn using the code blocks shown in the illustration below.

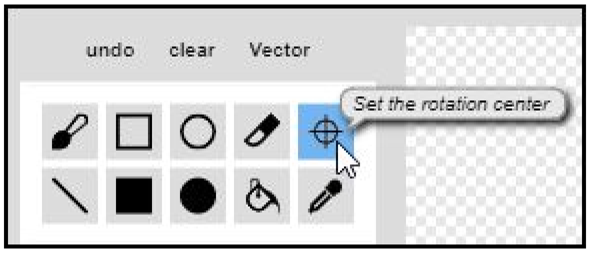


```
pen down
set pen size to 4
move 100 steps
```

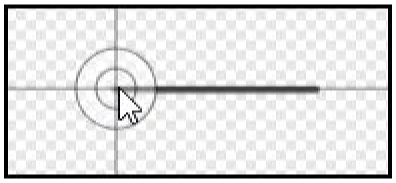
The line can be converted into a costume by right-clicking the drawn line and selecting the *Pen Trails* option in the dialog box that appears.


edit
show all
pic...
pen trails
turn all pen trails and stamps into a new costume for the currently selected sprite

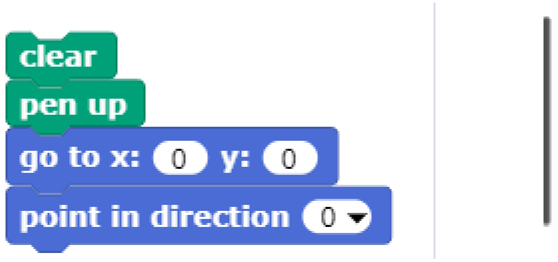
The rotation center of the line converted to a costume should be set to the origin of the line. This can be accomplished by selecting the *cross-hair icon* in the Paint Editor.



This option enables access to a cross-hair that indicates the rotation point. This cross-hair can be dragged to the origin of the line as shown in the illustration below.

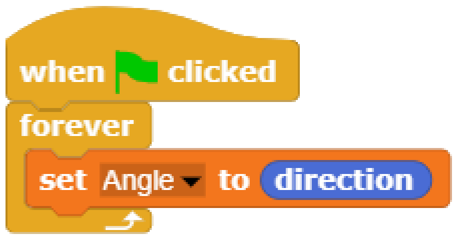


The sprite with the costume in the shape of a line should then be positioned in the center of the screen pointing upward. (The **Clear** code block is used to clear any lines that are drawn on the stage.)

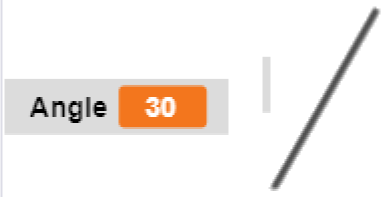


```
clear
pen up
go to x: 0 y: 0
point in direction 0
```

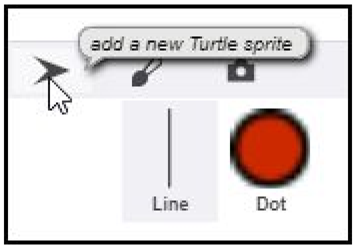
The angle in which the line is pointed can now be determined through creation of a variable named *Angle*. The value of this variable is set to the **Direction** in which the line is pointing.



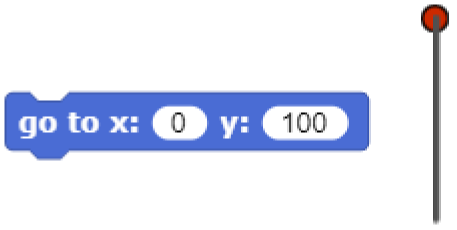
```
when green flag clicked
forever
  set Angle to direction
```



The sine function can be derived by determining the vertical height (i.e., Y coordinate) of a point on the end of the line. The Y coordinate can be determined by creating a second sprite (called *Dot* in the example below) and placing it on the end of the line.

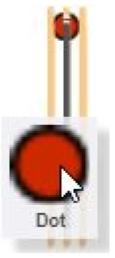


The second sprite, named Dot, can be placed on the end of the line through execution of the code block **Go to X: 0 Y: 100**. (Note: this code block must be placed in the script area of the Dot sprite *and not* in the script area of the Line sprite.) The Dot sprite should now be positioned on the end of the Line sprite.

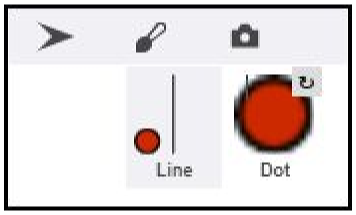


```
go to x: 0 y: 100
```

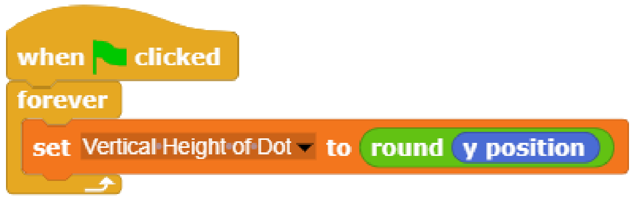
To group the Dot sprite with the Line sprite so that they move together as the line is rotated, the icon of the Dot sprite (in the sprite corral beneath the stage) should be dragged on top of the Line sprite. When the icon is superimposed over the Line sprite, the halo consisting of a tan line will surround the line sprite. This indicates that the two sprites have now been grouped so that they will move together.



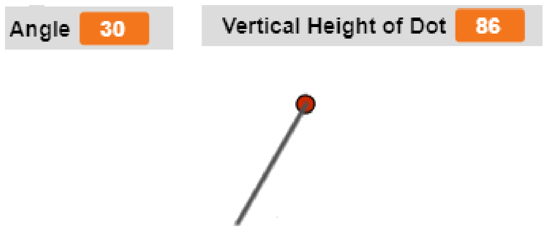
After the Line sprite and the Dot sprite have been grouped together, the icon of the Line sprite will change, with an image of the dot beside it, to indicate that the two sprites are now grouped.



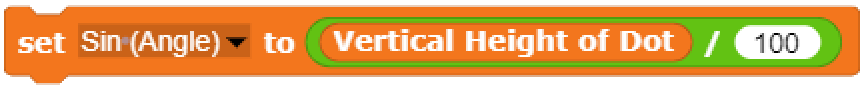
The Dot sprite can now be used to report the vertical height of the point on the end of the line by setting a variable named *Vertical Height of Dot* to the Y Position of the Dot sprite.



```
when [flag] clicked
forever
  set Vertical Height of Dot to round y position
```

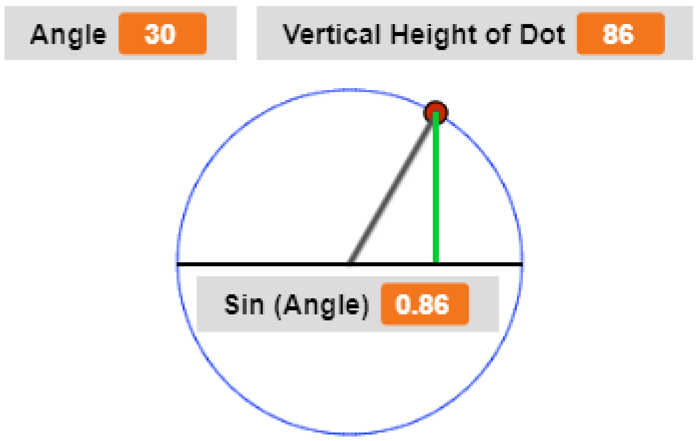


The sine function can be derived by dividing the vertical height of a point on the circumference of a circle (shown in green in the illustration below) by the length of a line extending from the center of the circle to the edge of the circle (i.e., the radius). Since the length of the line created is 100 steps, the sine of the angle can be calculated by dividing the vertical height of the dot at the end of the line by 100.



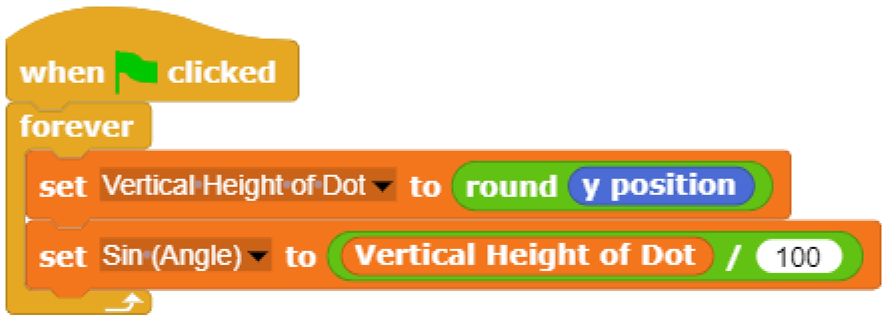
```
set Sin (Angle) to Vertical Height of Dot / 100
```

Consequently, the sine of 30 degrees is “0.86”.



Selecting a line length of 100 simplifies the calculations, since it means that the value of the sine value will always be the vertical height of the dot on the edge of the circle divided by 100. For example, if the vertical height of the dot is 70, the corresponding sine value will be “0.7”.

The sine value can be calculated for any angle by updating the procedure for the Line sprite, calculating the sine value as the Line sprite rotates.



```
when [flag] clicked
forever
    set Vertical Height of Dot to (round (y position))
    set Sin (Angle) to ((Vertical Height of Dot) / 100)
```

Exploration 2.8 Reconstructing the Sine Function. Use the procedures developed in this section to determine the value of the sine function for the following angles: 0, 15, 30, 45, 60, 75, and 90 degrees.