

Physics Calculations and Simulation for a Mangonel-style Rubber Band Catapult

A Mangonel uses the potential energy created by tension in an elastic band, or torsion in a twisted rope. We'll be performing the calculations for the former, which is the more common approach in rubber band catapults.

Potential Energy of the Rubber Band

Hooke's law tells us that the force needed to stretch a spring (or in our case, a rubber band) is:

$$(1) F = -kx$$

Where x is how far the spring is stretched, and k is a "spring constant" which is unique to each type of spring – we must determine this experimentally. Thus, we can rearrange our equation to:

$$(2) k = -\frac{F}{x}$$

From this equation we can find out the spring constant k by stretching our rubber band to x and measuring the force F it generates. Once we have our k value, we can turn our attention to the potential energy PE_{spring} stored in the stretched rubber band – energy that can be converted to work by our catapult. This quantity of energy (measured in Joules) can be calculated by the equation:

$$(3) PE_{spring} = \frac{1}{2}kx^2$$

The resulting potential energy PE_{spring} is the total amount of energy our catapult assembly has available to launch its projectile. Next, we need to consider how the launch actually occurs.

Rotational Kinetic Energy of the Catapult Launch Arm

The catapult launch mechanism consists of a launch arm, anchored to a hinge or axle. The Potential Energy of the spring is converted into motion of this arm, which swings around that axle. The projectile itself is put into a basket on the end of the launch arm – thus our potential energy is converted into rotational kinetic energy $KE_{rotation}$:

$$(4) PE_{spring} = KE_{rotation}$$

This, of course, assumes no energy is lost in the conversion; in reality some energy will always be lost – converted to heat by friction, or absorbed by our joint materials. But this amount is small enough that we can ignore it for the purposes of our calculation. The kinetic energy of rotation can be expressed by the equation:

$$(5) KE_{rotation} = \frac{1}{2}I\omega^2$$

In this equation, ω is the angular velocity – it measures how quickly our catapult arm spins around its axle. I is the moment of inertia. For our launch arm, this is the same as a rod rotating on its end, or:

$$(6) I = \frac{1}{3}mr^2$$

In this equation, r is the length (radius) of our launch arm, and m is the mass of the projectile. We can substitute the moment of inertia (equation 6) into our kinetic energy of rotation equation (equation 5), and obtain:

$$(7) KE_{rotation} = \frac{1}{2}\frac{1}{3}mr^2\omega^2 = \frac{1}{6}mr^2\omega^2$$

Angular Velocity of the Projectile

While cupped in the basket at the end of the catapult's throw arm, the acceleration of the launch arm creates angular velocity ω , until the launch arm's motion is arrested by the stopping brace. At this point, the projectile is freed to move on its own, and this angular velocity is transferred into linear velocity v , tangential to the arc that the projectile was traveling at up to that point.

Therefore we need to calculate ω , which can be done by substituting equations (7) and (3) into equation (4):

$$(8) \frac{1}{2}kx^2 = \frac{1}{6}mr^2\omega^2$$

And solving for ω :

$$(9) \frac{1}{6}mr^2\omega^2 = \frac{1}{2}kx^2$$

$$(10) \omega^2 = \frac{6kx^2}{2mr^2} = \frac{3kx^2}{mr^2}$$

$$(11) \omega = \sqrt{\frac{3kx^2}{mr^2}}$$

Finally, we can convert the angular velocity to linear velocity v with the equation:

$$(12) v = \omega r$$

Substituting our value of ω from equation 12, we obtain:

$$(13) v = \sqrt{\frac{3kx^2}{mr^2}}r = r \frac{x}{r} \sqrt{\frac{3k}{m}} = x \sqrt{\frac{3k}{m}}$$

Notice that the radius of the launch arm r disappears from our equation as we simplify it. This suggests that the launch velocity of the projectile is not affected by the length of the arm, only by the mass of the projectile, the extent to which the spring is pulled back x , and the spring constant k . Why does this happen? How does varying the length of the throw arm change the path and velocity of the projectile

while it is cupped within the launch arm's basket? Do we need to consider the launch arm's length in designing our catapult? Why or why not?

Equations of Ballistic Motion

Once released from the catapult, the projectile is only affected by gravity, which exerts a constant downward force upon it (it also must contend with friction from the air molecules around it – air resistance – and any wind, but we'll ignore these for now). Thus, we can use Newton's equations of ballistic motion to calculate our projectile's position at a set time after launch t :

$$(14) x = v_0 t \cos(\theta)$$

$$(15) y = v_0 t \sin(\theta) - \frac{1}{2} g t^2$$

Notice we split the initial velocity into a vertical (y) and horizontal (x) component. The *sine* and *cosine* functions allow us to determine how much of our initial velocity contributes to each of these directions, given the angle between the direction our projectile was launched and the horizontal is θ . The angle θ is our angle of release, determined by where in our launch arm's rotation it slams into the stop brace.

Also, we have an extra term in our equation for vertical motion: $-\frac{1}{2} g t^2$. This is the downward drag exerted by gravity, g (which we typically approximate with 9.8 m/s^2). As the value of time is squared (t^2), we know the effect of gravity on our projectile will increase over time, pulling it downward. Interestingly, we have no such term in our x -position, which means our projectile will continue to fly forward at the speed we launch it, until it strikes something (like a castle wall, or the ground).

Simulating our Projectile Motion in Scratch

We now have enough information to simulate our ballistic path over time on the computer. To do this, we will utilize the programming environment, Scratch, developed by MIT (available at <http://scratch.mit.edu>). In addition to providing a drag-and-drop programming interface, Scratch has a built-in graphical component – a stage where Sprites can move and interact, based on the commands you program. Let's start with a projectile-shaped sprite. Open Scratch, click the "add new sprite from file" icon, and browse to the "Things" folder. In this folder, you should find several ball sprites – I'm using the beach ball, but any should work.

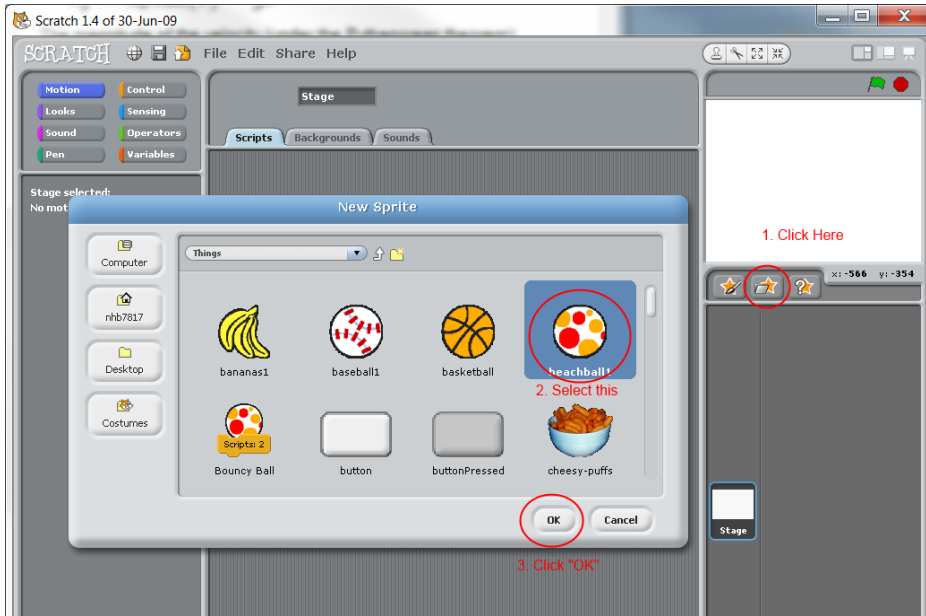


Figure 1 - Adding a projectile sprite to Scratch

Now we need to change the ball's position over time, according to our equations of motion. The `go to x: 0 y: 0` block offers one possible way of accomplishing this – we simply need to supply it with an x and y position. We can use equations (14) and (15) to calculate these values, creating variables to represent x , y , v_0 , t , and θ :

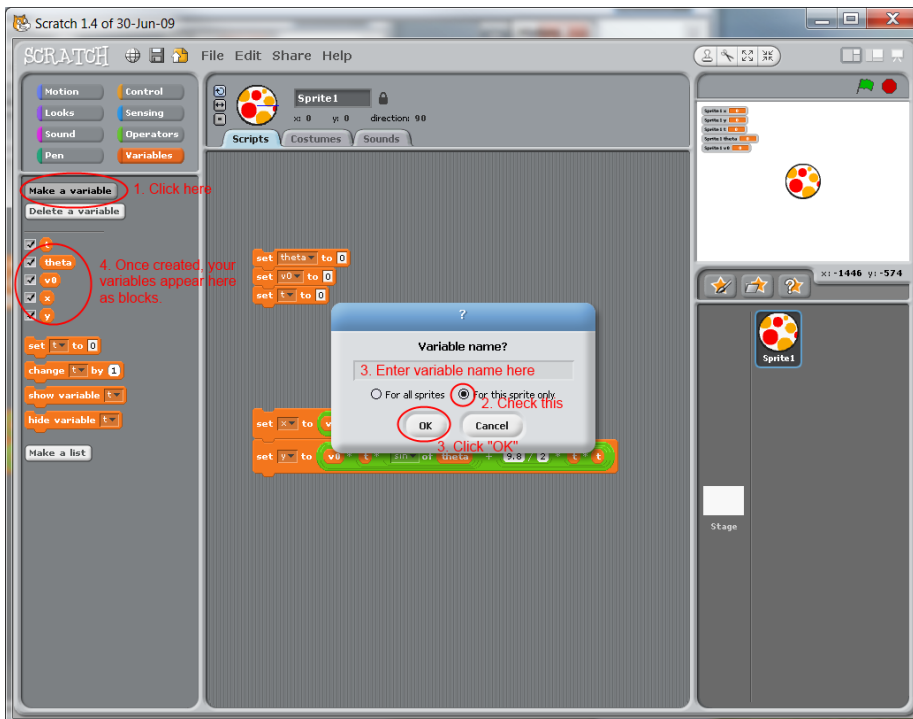




Figure 2 - Adding variables to Scratch

Once we've created the variables, we use the  blocks to set the values of x and y , calculated using equations (14) and (15):



Note that in Scratch, order of operations is implicit in the nesting of blocks – i.e. in our first equation, $\cos(\theta)$ is calculated first, then multiplied by t , and then by v_0 . Also notice Scratch does not have a “squared” block, so we use the mathematically equivalent . We could have also pre-calculated $9.8/2$ and used that value rather than the operator block .


Now we need to calculate the value of v_0 , which can be done with equation (13). This equation adds several more variables to our program: x , k , and m . But we already have a variable x , which is the horizontal position of our projectile! So let's rename our variable x from equation (13) to *stretch*, as it is the “stretch” of the rubber band. Then we can add our new variables like we did before, and then calculate the value of v_0 :



Now we need to provide values for the other variables: m , k , *stretch* and θ . You should obtain m by measuring the mass of your projectile (I'll use 2.7g, or 0.0027kg, the weight of a ping-pong ball). You should have obtained your k value experimentally from stretching rubber bands (I'll use 3.16 in my example). The values of *stretch* and θ depend on your design. For now, let's use 10cm (0.01m) for *stretch* and 45° for θ .

But what about time t ? We want to see our ball move along its trajectory, which means we actually want to be *changing* the value of t as our program is running, and move our ball each time. We can



accomplish this with a repeat loop  which repeats any instructions contained within it multiple times (the number specified after the word “repeat”). Thus, we can calculate our x and y , move the ball to that position, and change our time t by a set value inside the repeat block, and it will be repeated over and over. The amount of time we change t by is known as our *simulation time step*. The smaller the timestep, the more detailed our simulation will be (and the longer it will take to run). I'll use a time step of 0.01s (1/100 of a second) in my simulation:

```

repeat 1000
  set x to v0 * t * cos of theta
  set y to v0 * t * sin of theta - 9.8 / 2 * t * t
  go to x: x y: y
  change t by 0.01

```

Thus, in my example, we'll run 1000 iterations of the simulation, with a timestep of 0.01 (each iteration will calculate and position the ball 0.01s later than the last iteration).

Note: if your ball moves too fast, you can add a `wait 0.01 secs` block within the repeat loop. This will make each iteration pause for a fraction of a second, slowing down the simulation.

Your current Scratch program should look something like this, and you can run it by clicking the green flag in the corner:

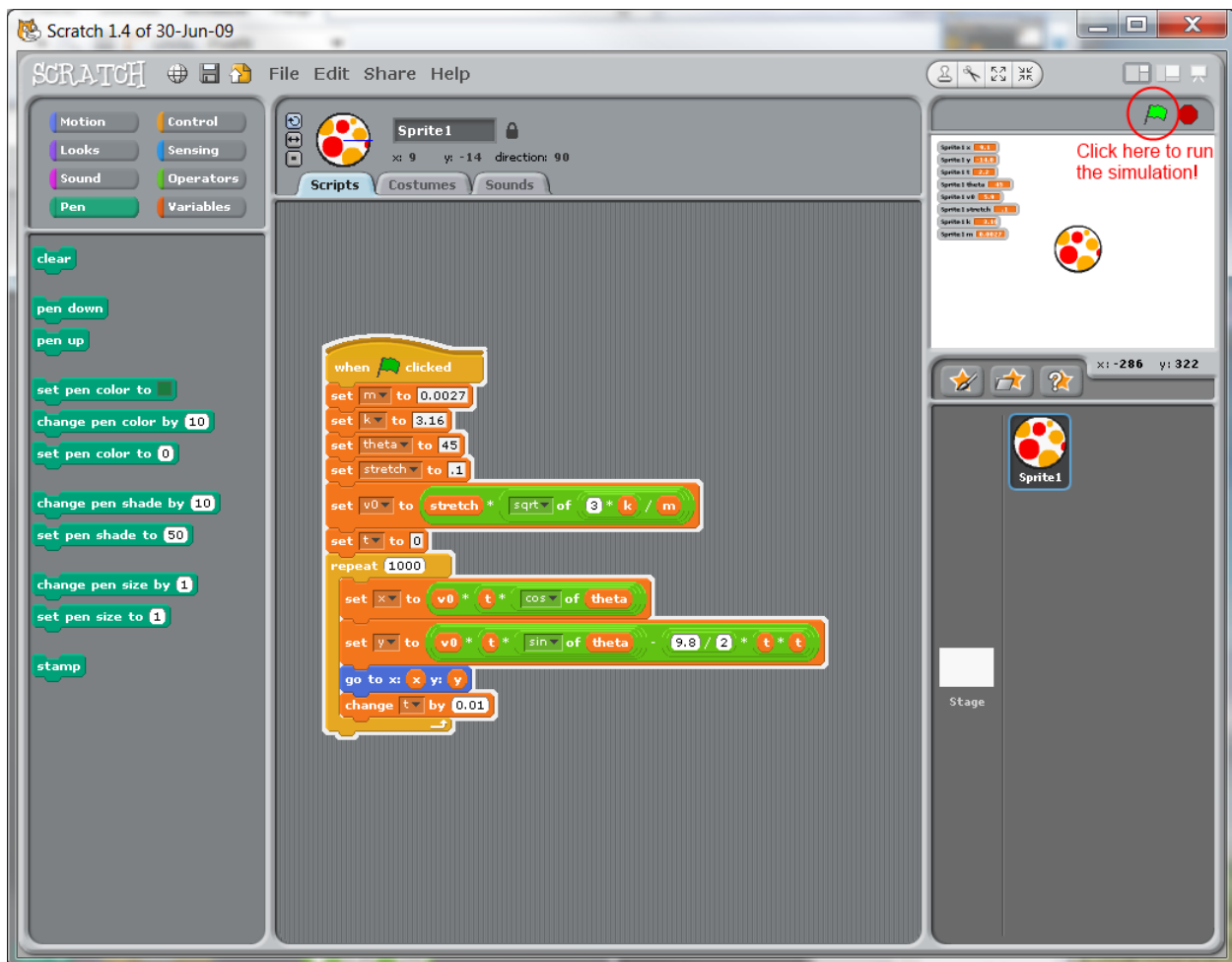



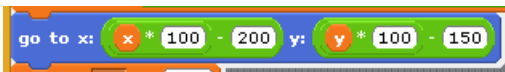
Figure 3 - Simulation of catapult projectile motion in Scratch

Improving the Visualization

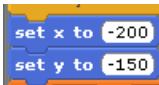
When you run the simulation, you'll likely notice a few things. First, your ball won't move very far at all.





This is because the  is moving to position (x,y) measured in pixels on the screen (which are very small). Your calculations are calculating the position in meters – so a single meter of distance corresponds to a single pixel on the screen! This is very small, and makes it hard to see the motion of our ball. Second, your ball starts in the center of the screen. Likely we'll want to actually have it start somewhere near the bottom-right corner, so we can see the full path of motion.

We can accomplish both of these goals by *transforming* our coordinate system. We can shift the ball's position to the corner by *translating* it, or subtracting a constant value from x and y (I'll use 200 and 150 pixels). Further, we can “zoom in” on our moving ball by *scaling* the calculated value of x and y. Instead of 1 pixel = 1 meter, we can use a bigger ratio, like 100 pixels = 1 meter (or 1 pixel = 1 cm). Thus, we take our calculated x & y and multiply them by 100. Combining these two transformations gives us:



We will also want to set our ball's initial position to (0,0) – which in our new coordinate system, is (-200,-150) pixels. We'll do this at the top of our sprite's script with:



Now if we run the simulation, we can see the ball much more clearly. But it would also be nice to plot the movement of the ball in such a way that we can see the full range of movement. Scratch includes a “pen” that we can use to draw the motion of the sprite onto the background – commands for using it are in the “pen” menu. In particular, we'll be interested in the  and  blocks, which tell Scratch to start and stop drawing. If we place a  before our repeat loop, and a  after it, we'll graph the ball sprite's motion as it moves across the screen (we need to pick up our pen after we finish, or the next time we run the simulation it will draw a line back to the beginning).

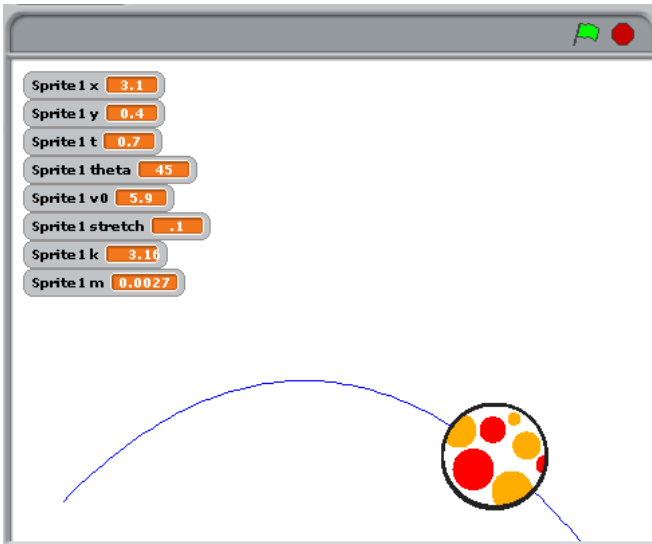


Figure 4 - Plotting the projectile path in Scratch

Adding a Grid

Now we can see the path of the ball, but it would be even better if it was a proper graph, with lines denoting units of measure. There are actually a couple of ways to do this in Scratch – we could create an image of a graph grid, with axes and labels, and use it as a background, or we can tell Scratch to draw them for us.

I'll walk you through the latter – we'll start by creating another sprite. What it looks like won't matter; we just need it to draw for us. In fact, we can make our first step hiding the sprite with the `hide` block. We'll also want to clear any pen drawings on the screen with `clear`, and change the pen's color so it isn't the same as our projectile's path with `set pen color to`.

Now we're ready to draw a grid. We'll start with the vertical lines – let's say we want a vertical line every ten pixels from 0 to 400 pixels. Each of these lines will correspond to a 10 centimeters (remember we scaled our pixels to 1pixel = 1cm). We can draw a single line by moving our pen to the start of the line, putting it down, and then moving to the end, and then picking it up:

```

go to x: -200 y: 150
pen down
go to x: -200 y: -150
pen up

```

We want to repeat this for each vertical line. We could enter the same command 41 times, or... we could use a repeat block to repeat the command that many times:


```
set x to 0
repeat 41
  go to x: x - 200 y: 150
  pen down
  go to x: x - 200 y: -150
  pen up
  set pen size to 1
  change x by 10
```

Note that we use the variable x to remember where the last line was placed – every time we repeat our commands, we move the next line over by 10 pixels. The resulting screen will look like this:

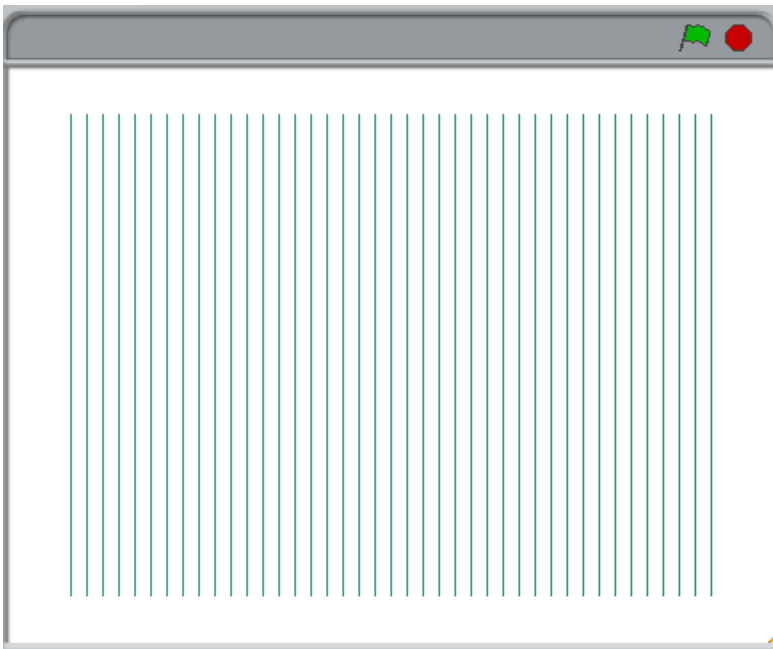


Figure 5 - Vertical grid lines in Scratch

We'll want to do the same thing for our horizontal lines:

```
set y to 0
repeat 31
  go to x: -200 y: y - 150
  pen down
  go to x: 200 y: y - 150
  change y by 10
  pen up
broadcast grid drawn
```

And the resulting grid will look like:

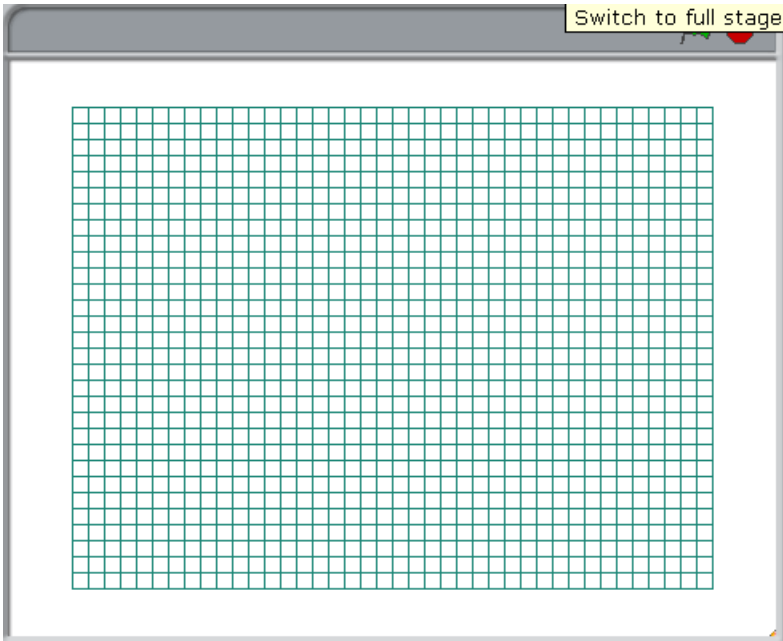


Figure 6 - Vertical and horizontal grid lines in Scratch.

We can further refine our grid by darkening every ten lines – corresponding to each meter. To do this, we want to make our pen thicker (which makes the line darker). This can be done with the

`set pen size to 2` block. A pen size of 2 will draw lines 2 pixels thick. We can then draw the centimeter lines using the same strategy we did before, but 100 pixels apart:

```
set x to 0
repeat 5
  go to x: x - 200 y: 150
  pen down
  go to x: x - 200 y: -150
  change x by 100
  pen up
set y to 0
repeat 4
  go to x: -200 y: y - 150
  pen down
  go to x: 200 y: y - 150
  change y by 100
  pen up
```

Our refined grid looks like this:

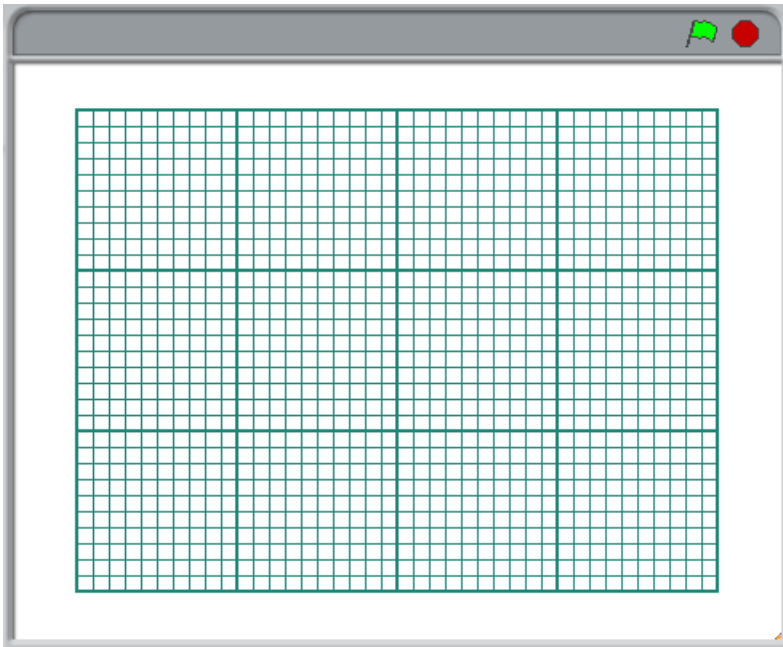


Figure 7 - Grid lines with darkened meter marks

Now, we would also like to have labels on our graph – but Scratch doesn't have an easy feature for writing text onto the screen (we could draw it with the pen, or use sprites for our letters, but both take some serious work). An easier way would be to turn what we just drew into an image, and add our labels to that image, and then use the edited image as a background on the stage. We can do this by right-clicking the stage and choosing "save a picture of the stage".

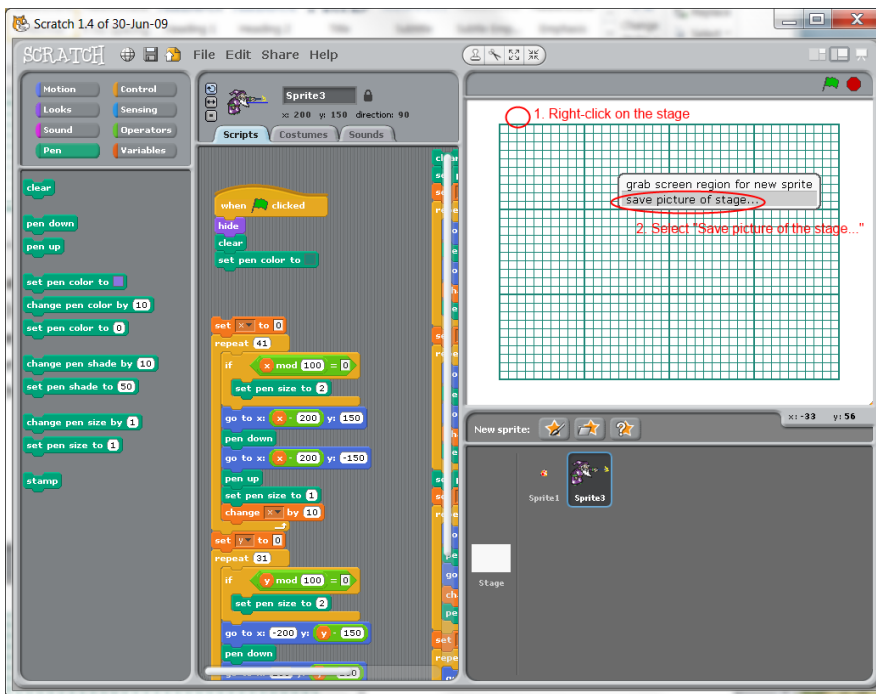


Figure 8 - Saving the Scratch stage as an image

We can then open the image file in an image editor and add labels and any other details we would like to have:

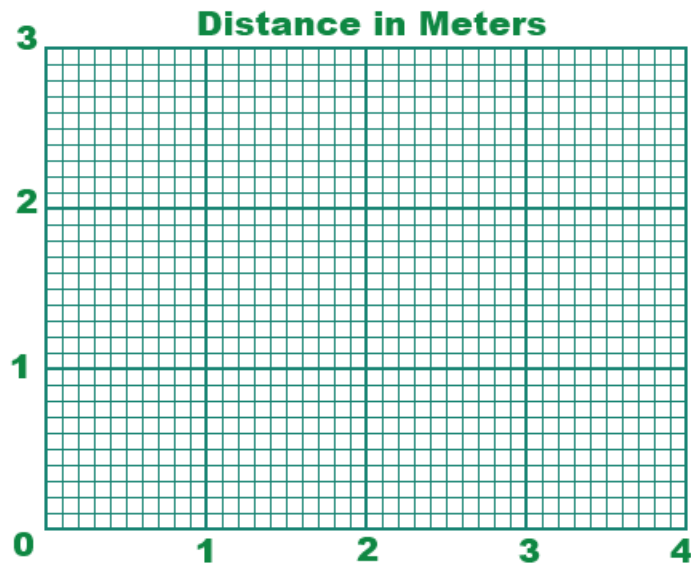


Figure 9 - The graph with labels added using Paint

Then switch back to Scratch, select the Stage, choose its "background" tab, and import the picture you just saved:

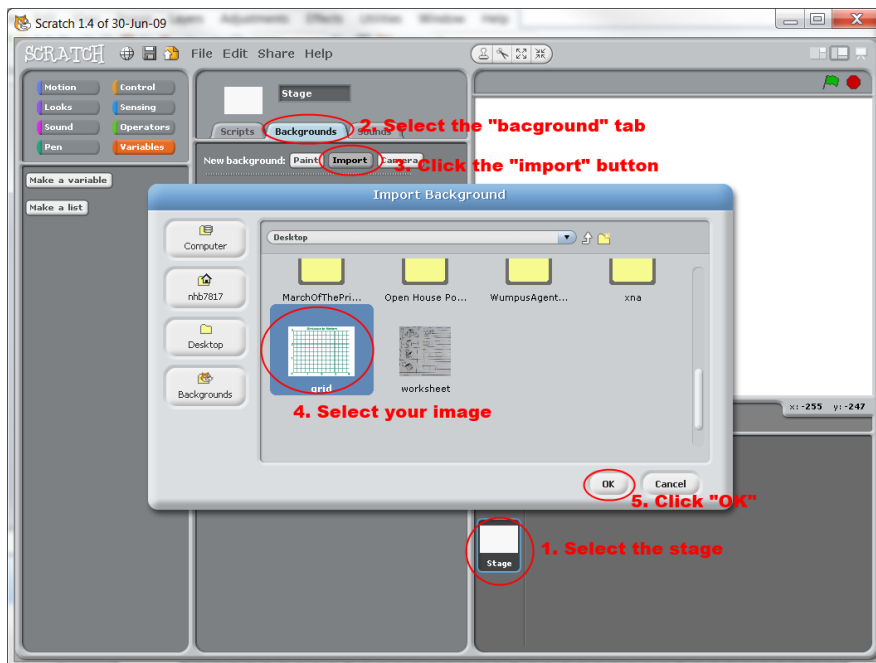
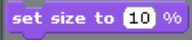


Figure 10 - Importing the labeled grid as a background

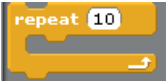
And now we have a background with a scale! One final tweak we may want to make – our projectile is quite large. We can use the  block to make it fit better within our grid.

Finding an Optimal Design

Now that we have a complete simulation, we can discover how well different catapult designs might work by tweaking the release angle θ , the extension of the rubber band $stretch$, and the mass of the projectile m . And while our arm length r fell out of our projectile equations, it still has a role to play in how much $stretch$ we can achieve.

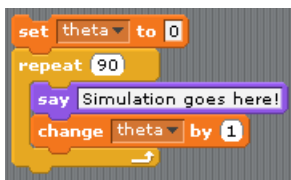
But before we can find an optimal design, we need to first identify what we mean by “optimal.” Is an optimal catapult one that throws our projectile the farthest, or one that does the most damage when it hits? If we were doing pumpkin chuckin’, distance is the goal. But if we’re pounding on the walls of a castle under siege, damage would be. Different goals often require very different design decisions.

Let’s say we are going for optimal distance – in that case we want to maximize the x values we can achieve. We could just start plugging new numbers in for each of our values, run our simulation, and then try another value. But as engineers, we want to be *systematic* about how we test. For example, we might start with an angle of 0 degrees (launch is completely horizontal) and try every full degree until we reach 90. Doing this by hand would be quite exhaustive. But we’ve already seen a block that

can help us repeat a task - . Let’s try using that block to solve this question. We’d want to:

1. Start with an angle of 0
2. Run our simulation
3. Increase the angle by 1 degree
4. Jump back to step 2 and repeat

What we just wrote is an *algorithm* – a step-by-step process for solving a problem. We can express this using Scratch blocks:



For clarity, I omitted the actual simulation algorithm (the purple block is a stand-in for it). But if we were to incorporate our new algorithm into our simulation code, it would look like:

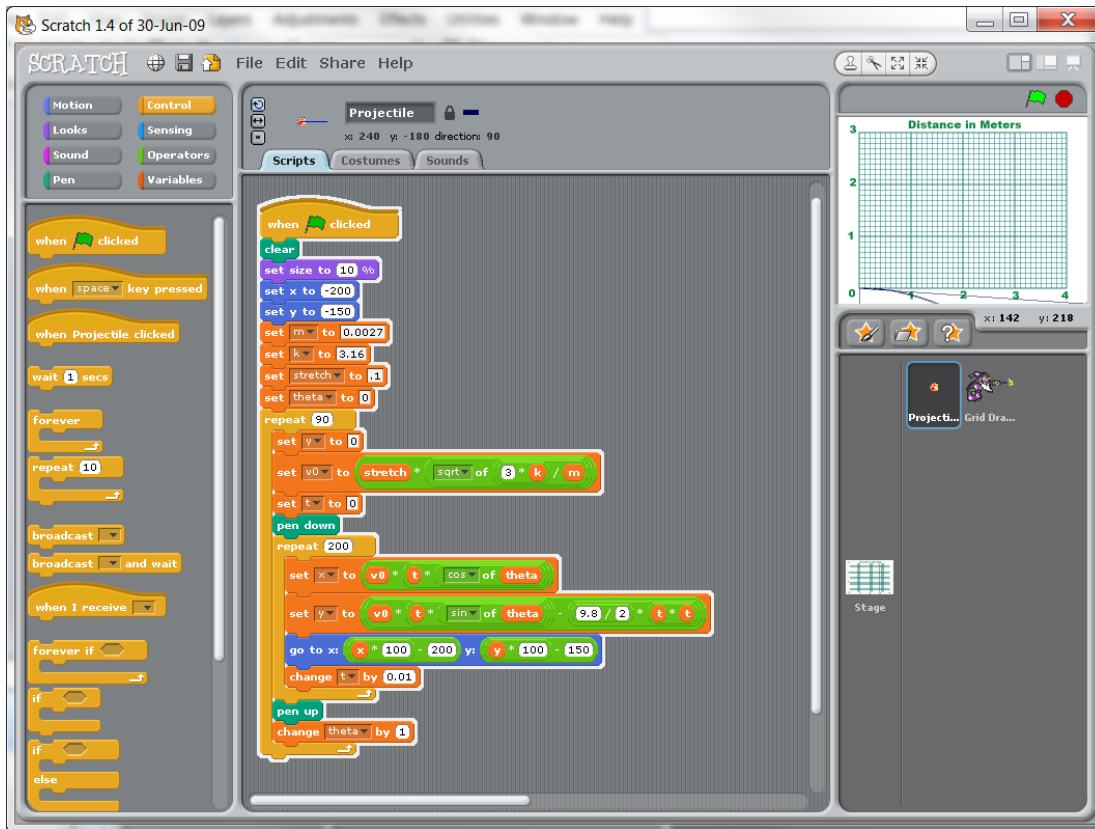


Figure 11 - Running the projectile simulation to create a graph

And our graph would look like:

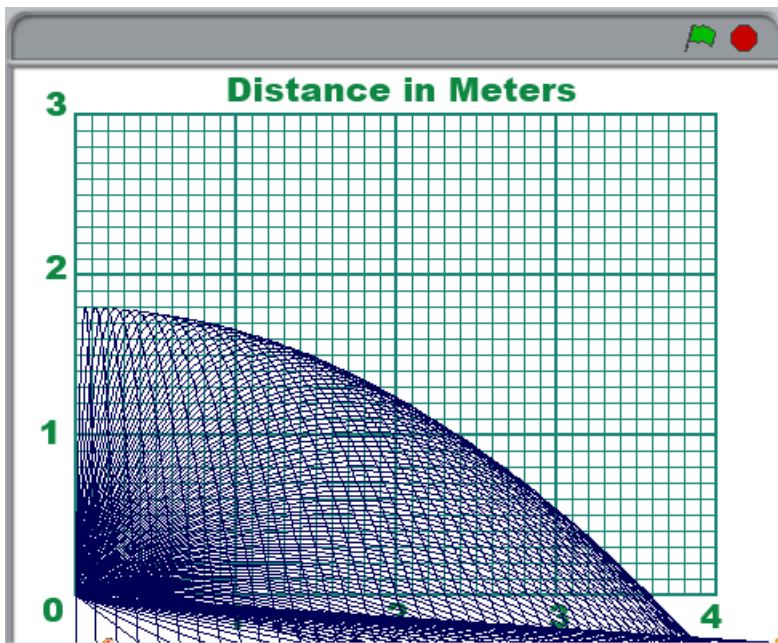

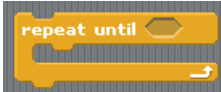
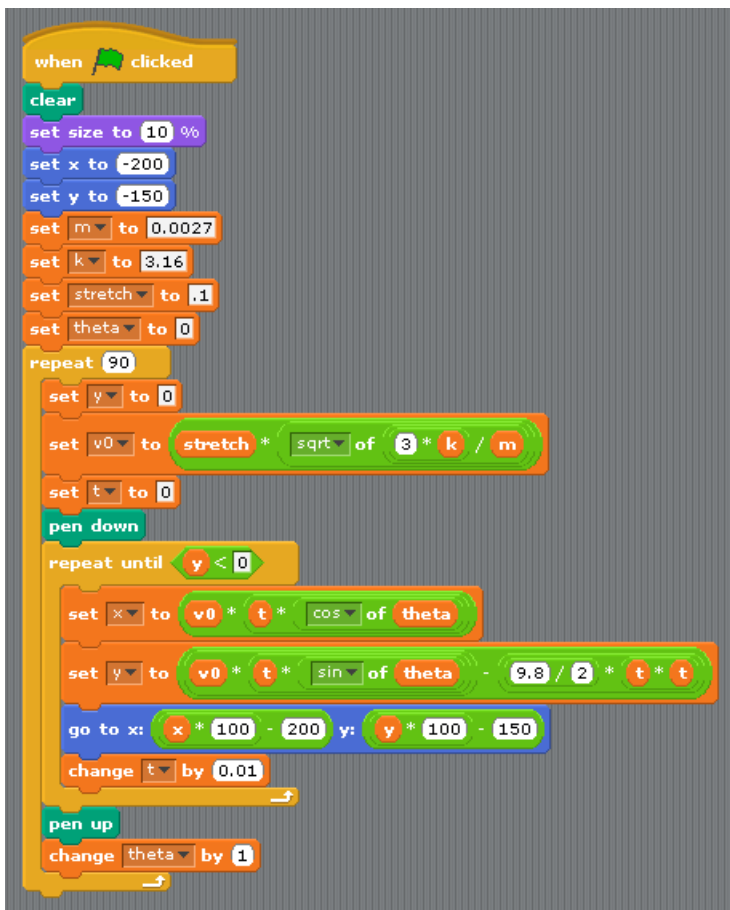


Figure 12 - The graph generated by the projectile simulation

There are a couple of things to notice about our graph – first, our graph continues even after our height falls to 0 (i.e. we’ve hit the ground). It’s not a huge issue, but it’s messy. Second, it is very hard to tell one launch from another – perhaps we can find some way of making each line more distinct. Finally, this graph suggests that our maximal distance, given the stretch and mass, would fall close to 3.6 m. But what is the angle that we used to reach that? Let’s tackle these issues one at a time.

First, to avoid drawing after we’ve hit the ground, we need to stop repeating our simulation once we’ve reached the ground. This happens when our $y = 0$. However, $y = 0$ when we first launch, and unless our timestep is sufficiently small, it might actually go from $y = 0.1$ to $y = -0.4$, and never actually equal 0! We can get around both of these issues by checking if $y < 0$ – which will only happen *after* the projectile hits the ground. In Scratch, this is the Boolean comparison . We can use this as the test of a

conditional statement, like a . This block repeats the instructions within it until the test (the diamond-shaped block) is “true”. By combining it with our Boolean test, and substituting it for our regular repeat we can run our simulation until the projectile hits the ground:



```

when clicked
  clear
  set size to 10 %%
  set x to -200
  set y to -150
  set m to 0.0027
  set k to 3.16
  set stretch to .1
  set theta to 0
  repeat 90
    set y to 0
    set v0 to stretch * sqrt of 3 * k / m
    set t to 0
    pen down
    repeat until y < 0
      set x to v0 * t * cos of theta
      set y to v0 * t * sin of theta - 9.8 / 2 * t * t
      go to x: x * 100 - 200 y: y * 100 - 150
      change t by 0.01
    pen up
    change theta by 1
  
```

Also, note it was necessary to reset y to 0 before the simulation loop – otherwise our y is still negative from the last simulation! The resulting graph is much neater, and it runs faster, as we aren’t bothering to simulate any time after the projectile hits the ground:

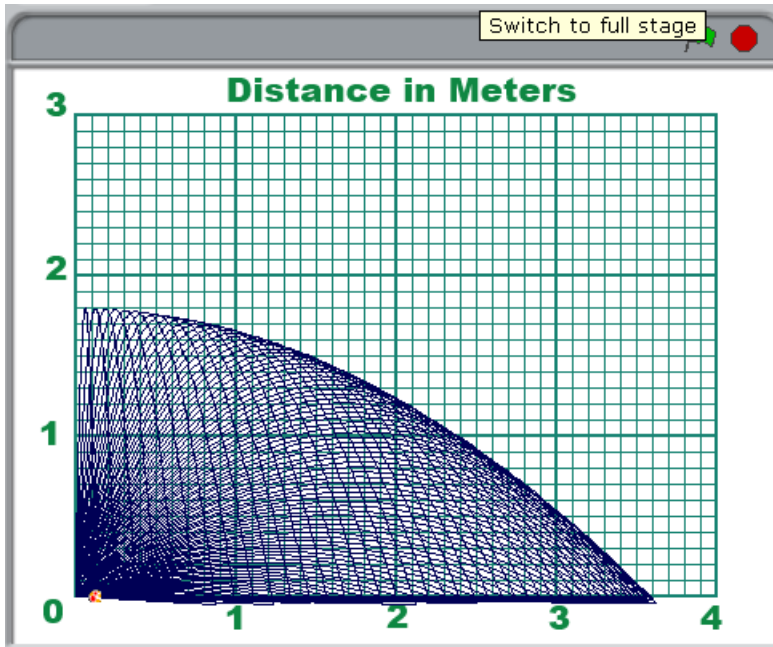


Figure 13 - The graph produced by the simulation when considering impact with the ground

Second, to distinguish the graph lines, we can use a slightly different pen color each time with a `change pen shade by 10` block. This can be added to the simulation where we put the pen down, and then each simulation run will use a slightly different shade of color.

Third, what we are really asking for here is the value of θ when the distance traveled is maximized.



Remember, we end our `repeat until` block *right after* the projectile strikes the ground. This means that at that point, our x is at the distance that projectile traveled. So before we do our next simulation, we can test if this is the maximal distance. An algorithm for doing this might be:

1. Set the maximal distance to 0 (as we haven't run any simulations yet)
2. Set θ to 0
3. Run the simulation
4. If the simulation's x is greater than the maximal distance, set the maximal distance to x and save θ as our "ideal" θ
5. Increase θ by 1
6. Repeat steps 3-6 until θ is 90 degrees

Notice this algorithm incorporates our earlier algorithms – the one for changing θ , and the one for running the simulation. To use it in scratch, we need two new variables: *maximal distance* and *ideal*

θ , and we need a conditional `if` block. The `if` block only runs the code inside of itself if the test in the diamond-shaped socket is true (i.e. $x > \text{maximal distance}$). The Scratch version will look like this:


```

set ideal theta to 0
set maximal distance to 0
repeat until y < 0
  say Run the simulation!
  change t by 1
if x > maximal distance
  set ideal theta to theta
  set maximal distance to x

```

Note – again for clarity I have omitted the full simulation algorithm and used the purple block as a placeholder. Our final algorithm therefore looks something like this:

The screenshot shows the Scratch 1.4 interface with a script for a projectile simulation. The script is as follows:

```

when green flag clicked
  clear
  set pen size to 10 %
  set x to -200
  set y to -150
  set m to 0.0027
  set k to 3.18
  set stretch to 11
  set theta to 0
  set ideal theta to 0
  set maximal distance to 0
  repeat 90
    set v to 0
    set v0 to stretch * sqrt of 3 * k / m
    set t to 0
    change pen shade by 10
    pen down
    repeat until y < 0
      set x to v0 * t * cos of theta
      set y to v0 * t * sin of theta - 9.8 / 2 * t * t
      go to x: x * 100 - 200 y: y * 100 - 150
      change t by 0.01
    if x > maximal distance
      set ideal theta to theta
      set maximal distance to x
    pen up
    change theta by 1

```

The graph on the right shows the projectile's path, with the x-axis labeled 'Projectile maximal distance' and the y-axis labeled 'Projectile theta in cm'. The path is a blue curve starting at (-200, -150) and ending at (0, 0).

Figure 14 - The final sprite script for graphing the simulation and finding an optimal launch angle

And, as we run it we will see that our maximal distance and ideal theta values regularly change as our projectile travels farther and farther, until it no longer travels farther than the previous simulation:

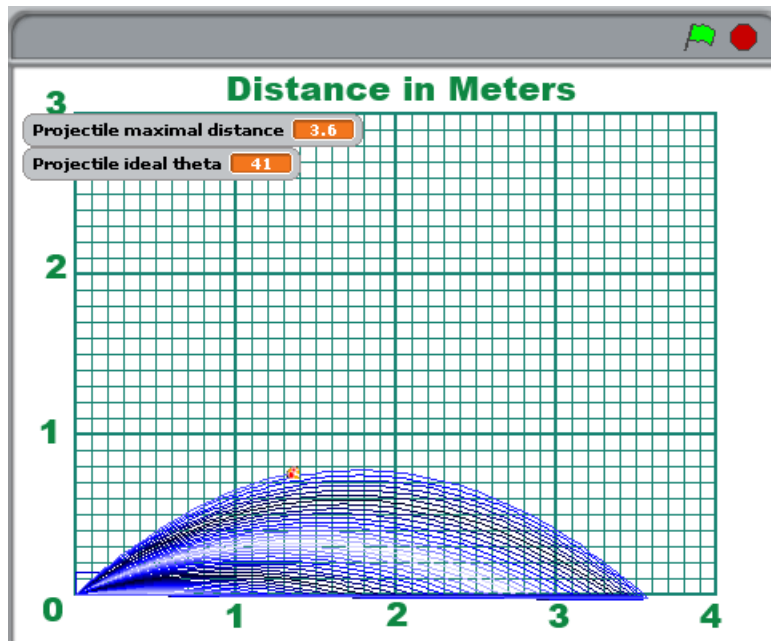


Figure 15 - The simulation running while calculating ideal launch angle

With the values I used for k and $stretch$, the optimal angle was 44 degrees, and it traveled a distance of 3.6 meters

Further Refinements

We can systematically vary more than one variable in our simulation to answer more complex questions, like “What is the optimal launch angle *and* stretch *and* projectile mass to achieve the greatest distance?” Remember too, the failure points of your materials (i.e. you won’t want to stretch a rubber band past its tolerance – or it will snap and your catapult will be ruined. Similarly, too heavy a projectile mass may snap your launch arm).

You can also calculate the forces your catapult’s structure needs to withstand – how much force the rubber band will exert on the catapult frame, for example, and how much force the launch arm will strike the stop brace with. Your choice of materials impose *constraints* on your design – even though a rubber band stretched 2m will give you an impressive launch simulation, unless you have a very strong rubber band it won’t happen in the real world.

Also, our simulation model ignores certain complicating factors like air resistance. Launching a ball of crumpled paper and a ping-pong ball of the same mass in our simulation as it stands will produce the same ballistic path, but in reality the creases of a crumpled paper ball make air resistance affect it far more strongly than a smooth ping-pong ball; the ping-pong ball will travel farther. You can incorporate the effects of air resistance and wind into the program as well to make it more true-to-life.