9.1 *Catch the ball (Pokemon)*

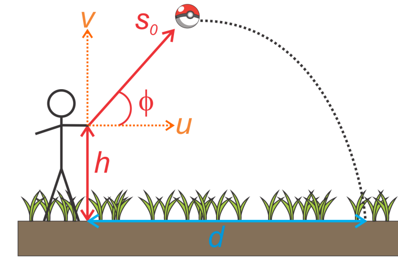

Figure 9.1: Schematic of the thrown-ball system.

In considering dynamic, time-stepping representations of physical (/biogeochemical) systems, we'll start with a simple, ballistics example – that of the trajectory of a thrown ball.

The system we'll consider is shown schematically in Figure 9.1. In essence: we want to determine d – the horizontal distance (m) that the ball travels before it hits the ground. The initial conditions are:

1. The ball is thrown from an initial height h (m).
2. The ball is thrown with an initial speed s_0 (ms^{-1}).
3. The ball is thrown at an initial angle θ with the horizontal.

We'll neglect any air desistence or spin imparter with the ball, and for the purpose of calculating its height, we'll ignore its diameter, i.e. we'll consider that the ball is level with the ground when its centre is at height zero. Over and above this, you'll only need to know the gravitational constant (i.e. gravitational acceleration) – $g = 9.81ms^{-1}$ (i.e. the ball is being thrown on an Earth-like planet near sealevel).

To simply things and the construction of the code and encapsulation of the physics of the model, we'll break it down into 4 steps:

Part I Considering only horizontal travel.

Part II Considering only vertical travel.

Part III Considering both horizonal and vertical travel and testing for when the ball hits the ground.

Part IV Add some graphical output.

Part I Start with a new m-file. Create a structure along the lines of Figure 9.2, i.e. you are going to need to define some constants (g), parameters (the initial height h, initial speed (s_0), initial angle (θ) of the ball).

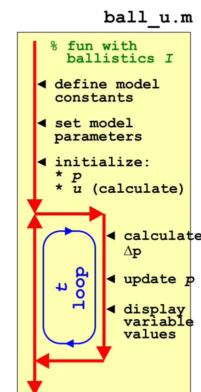

Figure 9.2: Schematic of the code for simulating the horizontal movement of a ball.

Because you are going to use a time-stepping approach (rather than solve the system analytically), you are going to need a loop in time, starting at time zero. Can you guess the time-step you need? No? Then we need to make the time-step a parameter that we can change to ensure that the system is solved well (i.e. accurately and without numerical instability). You could call this parameter e.g. dt and set it to an initial (guessed) value[1] such as $0.1s$. How long should you run the simulation for? This is also a sort of unknown at this point, at least until you have run the simulation a couple of times to get a feel for what the longest time the ball stays in the air might

[1] In the parameter section of the code.

be. So why not pick 100s to start with. Again, create a parameter to hold the value of the maximum model simulation time and assign its value in the parameter definition section of the code. Assuming a time-step parameter name of `dt` and a maximum time parameter, `max_t`, if your current time is called `t`, your loop structure will look like:

```
for t = 0:dt:max_t
    %SOME CODE
end
```

with time `t` starting at zero, and progressing to `max_t` in steps of `dt`.

What else do you need? You need a variable to represent the horizontal position of the ball (delineated here in the text as p, with units of m). This will start at zero and be updated within the loop. So also in the parameter section, why not define your horizontal position variable p and assign it a (initial) value of zero.

Lastly, you need to know the horizontal component of the balls' velocity.[2] You can calculate the (initial) horizontal component of velocity from the given initial conditions of initial speed (s_0) and initial angle of trajectory (θ). For now, pick any 'reasonable' values for s_0 [3] and θ [4]. In the figure, the velocity component is designated u.

Along with the schematic of the code structure, this should be all you need to create a basic code (but one at this point that does not actually 'do' anything). You should have a constant defined, and then 5 *parameters* – 3 representing the initial conditions of the model (the parts Figure 9.1 colored in red), plus 2 parameters for the maximum time and time step. You have 3 *variables* in the code so far – time t, which is part of the loop, (horizontal) position p, which you should have initialized to zero, and (horizontal) velocity component u, which you should have initialized calculated from s_0 and θ. There should be nothing in the loop so far.

Check that it runs without error even though it is doing nothing useful! Maybe add some debug (e.g. a line in the loop using `disp`) to check that the loop really does loop from zero to `max_t` in steps of `dt`.[5]

Now to add some code to the loop. In each time-step, i.e. each time around the loop, dt time (s) passes. In time dt, if the horizontal velocity of the ball is u, you should be able to calculate how far it moves, right? You need to add this increment in distance to the current value of the position variable p[6]. Do this.

Re-run the code. Check it works at all (if not: debug). Try adding debug code within the loop that displays the current time (t) plus value of p at each time-step, e.g.

```
for t = 0:dt:max_t
   %CODE TO UPDATE POSITION
```

[2] In the absence of air resistance, horizontal velocity does not actually change throughout the simulation (i.e. in each iteration of the loop, it will have the same value).

[3] On September 24, 2010, against the San Diego Padres, Chapman was clocked at 105.1 mph (169.1 km/h) – the fastest pitch ever recorded in Major League Baseball. If you convert 169.1 km/h into units of ms^{-1}, this will give you some reasonable upper limit for your initial thrown velocity.

[4] Obviously, the angle should lie between zero and 90 °(or else the throw is going backwards and/or into the ground). BE CAREFUL as **MATLAB** assumes that angles are in units of radians, so either work in units of radians throughout, or convert from degrees into radians when you calculate the velocity component based on the angle.

[5] Note that depending on whether or not `max_t` is divisible by `dt` with no remainder, your loop might not exactly finish at a value for `a` of `dt`.

[6] i.e. with code like

```
p = p + delta_p;
```

which you have seen endless times before now and should becoming wearily familiar ...

```
    disp(['current time = ', num2str(t), ', position = ', num2str(p)]);
end
```

so that you can track what is going on. (You can make a fancier output if you wish and add in the relevant units to the output.)

Strictly, when updating the position of the ball in the first iteration of the loop, time is dt at this point, not zero, which is what the loop thinks (you already have a position of zero at time zero – the initial conditions). So rather than starting the loop at zero, make a minor modification and start at a value of dt.

You should have a working model at this point, albeit only for the horizontal position of the ball.

Part II Now for tracking the vertical position (and velocity) of the ball. Copy your previous **m-file** and we can use this as a starting point for the new model.[7]

Think about what is different about the physics of the system (Figure 9.1) from before – this is going to directly inform how you adjust and add to the code. To start with, you should have noticed that the initial position (p) of the ball, does not start at zero, but rather at h. This is one change to make in the code (i.e. having defined h as a parameter, you subsequently use h to set the initial value of p). Also – the initial velocity component, v, is different from before (and in fact is assigned a different letter in Figure 9.1). So change the calculation of the initial velocity component and change the name of whatever variable you used for u to something distinct that you'll remember stands for v in the equation. Overall, the code structure looks like Figure 9.3.

You could, and indeed should, test the code so far. It should in fact do something very similar to before, with position p increasing, linearly, as a function of time (i.e. as the loop progresses in the number of iterations carried out). The only differences you should see are that p starts from value h and the rate at which p changes will be greater or less than before, depending on the value of θ you assumed.[8]

So far so good. Except balls generally do not continue travelling vertically for ever. You are missing gravity in this (vertical-only) model. Your variable for v (vertical velocity) now needs to change as a function of time and you'll need to update its value within the loop[9]. How are you going to update v? Well, the change in velocity with time is called acceleration and in this example the only force exerting any acceleration on the ball is gravity. Mathematically we can approximate the change in velocity, Δv as:

$$\Delta v = -\Delta t \cdot g$$

[7] So for instance we will now interpret p as the vertical, not horizontal position of the ball.

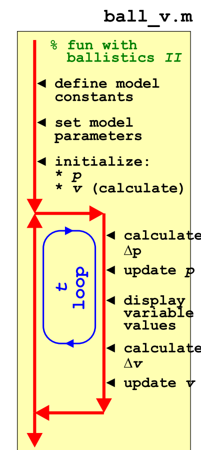

Figure 9.3: Schematic of the code for simulating the horizontal movement of a ball.

[8] What value of θ would result in an identical change in d with time (comparing the previous horizontal-only model with the new vertical (only) one)?

[9] Before or after the updating the position? Actually, a slightly tricky question.

where g is the acceleration due to gravity. Note the appearance of a minus sign in the equation if we are considering a coordinate system with distance upwards.

So in the loop[10] calculate the change in velocity during the time-step, and then update the value of v[11].

[10] HINT: at the end of the loop.

[11] Hint:

$$v_{(t+1)} = v_{(t)} + \Delta v$$

where $v_{(t+1)}$ is the new (at the next time-step) velocity and $v_{(t)}$ the current velocity

Re-run the model ... what happens? Does this seem 'reasonable' ... ? At this point you might consider whether you really do need to run the model for as long as $100s$. Play about with the assumed initial angle and also the velocity and get a feel for what is the longest the ball lasts in the air (i.e. until its position becomes negative).

Part III You should now have 2 working models (sperate **m-files**) – one for the horizontal position of the ball, and one for the vertical position (and vertical velocity) of the ball. You now want to combine the 2 sperate parts of the model. I suggest basing the combined model on the vertical model (as it is the more complicated of the 2) and hence copying-and-renaming the 2nd script.

How to merge? Mostly, the code content of the 2 individual models was almost identical. What you do need to copy across from the horizontal model is:

- The calculation of the initial value of u.
- The initialization of the horizontal position.
- The calculation of the change in horizontal position each time-step.
- The updating of the new horizontal position.

By now, you should have noted a slight problem – in both previous (sperate) models, the variable h was used to represent both horizontal AND vertical velocity. D'uh!

duh
exclamation informal
used to comment on an action perceived as foolish or stupid, or a statement perceived as obvious. As in:
"I used the same variable name twice – duh!"

My solution would be ... a vector to store the current position – just of one row and two columns, i.e. exactly as you might write a position in (x, y) notation. The horizontal position (x) is hence assigned the first element (`p(1)`) and the vertical position, the 2nd (`p(2)`). If you do this (i.e. resolve the variable clash this way), you'll need to edit how you set the initial conditions in the code, e.g.

```
p(1) = 0;
p(2) = h;
```

as well as how the position is updated in the loop. You can leave the name of the increment in position (Δp) the same if you wish (as this is a temporary variable whose value is replaced each time around the loop in any case).

Hopefully this works and runs ... Maybe add some output within the loop to track its progress, such as:

```
for t = 0:dt:max_t
    %CODE TO UPDATE POSITION
    disp(['(', num2str(p(1)), ',', num2str(p(2)), ') @ t = ', num2str(t)]);
    %CODE TO UPDATE VELOCITY
end
```

You should end up with output, depending on how you constructed the string to be displayed by `disp` (and what initial conditions you chose ...), like:

```
» ball_uv
(0.5,1.866) @ time 0.1
(1,2.634) @ time 0.2
(1.5,3.3038) @ time 0.3
(2,3.8755) @ time 0.4
(2.5,4.3491) @ time 0.5
(3,4.7247) @ time 0.6
(3.5,5.0021) @ time 0.7
(4,5.1814) @ time 0.8
(4.5,5.2626) @ time 0.9
(5,5.2458) @ time 1
(5.5,5.1308) @ time 1.1
(6,4.9177) @ time 1.2
(6.5,4.6065) @ time 1.3
(7,4.1973) @ time 1.4
(7.5,3.6899) @ time 1.5
(8,3.0844) @ time 1.6
(8.5,2.3808) @ time 1.7
(9,1.5792) @ time 1.8
(9.5,0.67938) @ time 1.9
(10,-0.31849) @ time 2
(10.5,-1.4145) @ time 2.1
...
...
```

which is far far far from exciting ... but does at least confirm a constant change in horizontal position with time, and a vertical position that initially increases above the initial condition ($h = 1.0$) but subsequently drops back and eventually falls below zero. And the point at which it reaches zero is the value of d of course.

The very least we could do at this point is to detect when the ball has reached the ground and terminate the loop. I'll leave this code for you to devise, but you'll need:

1. A conditional to test whether the vertical position has dropped below zero. This would go in the loop just after the position of the ball has been updated, And ...
2. The **MATLAB** command to exit a loop, which you have seen before.

Now you might note that when the ball reaches the ground (technically: its height falls below zero) and the loop exists, you may already be way below zero. In fact, if you are even the least little bit observant, you might note that the change in height per time-step at

the end of the simulation is quite large (order meter) and hence it is unlikely you'll ever capture the moment that the ball is very close to the ground. Unless you shorten the time-step, that is. So play about with a shorter time-step (you only need change the value you assigned to the parameter representing Δt in the code). How short does it have to be in order to catch the moment the ball reaches the ground (passes zero) to within e.g. $10cm$?[12] What about $1cm$?

[12] i.e. to have the loop terminate when the height is no more than $-10.0cm$.

Part IV Some graphics fun.

It would be kinda fun (really) to show the ball flying through the air. There are a variety of ways of doing this. We'll start with the simplest first and use `scatter`.

As a departure from previous plotting, we don't want to plot at the very end (after the loop)[13] but rather, plot each position as it is calculated, within the loop.

[13] Although if you stored the position of the ball at each time-step, you could re-play the trajectory afterwards.

First open a new graphics figure window and set `hold` on by adding the lines, before the loop starts:

```
figure;
hold on;
```

Within the loop, you want to plot each (x, y) position as it is calculated (after the position has been updated, that is):

```
scatter(p(1),p(2));
```

(feel free to add additional parameters to `scatter` to make the points smaller or larger, or filled, or whatever). Comment out any debug (`disp)` lines.

Well, not so exciting. The plots sort of appears all at once and there is no sense of animation or of the ball moving. **MATLAB** is just way too fast for its own good[14].

[14] This is a Trump-ism. In truth, **MATLAB** is about the slowest piece of *$&% about.

You can make the loop proceed slower, by adding a time delay – i.e. each time around the loop, **MATLAB** will take whatever time it needs to carry you the calculation and plot the current position PLUS whatever additional time you tell it to chill out for. The command is `pause` and you might initially try e.g.

```
pause(0.05);
```

which should insert a $50ms$ delay into the loop. Run it.

Now it has all got really trippy. If you tell it no different, `MATLAB` insists on auto-scaling the (x and y limits of the) plot. As the position of the ball increases (initially) in y-axis direction, and (constantly) along the *x-axis* direction, **MATLAB** periodically re-scales the axes. Annoying. So before the loop and after you create the figure window, why not prescribe axes limits(?) Having played with the model you

`pause`

MATLAB says: "*`pause(mjs)` pauses the MATLAB job scheduler's queue so that jobs waiting in the queued state will not run.*"

Garbage.

`pause(n)` will pause the execution of the code by `n` seconds.

`axis`

For once, helpfully, MATLAB says: "*`axis([xmin xmax ymin ymax])` sets the limits for the x- and y-axis of the current axes.*"
which is about all you need to know (other than the minimum and maximum limits along the *x-axis* are represented by `xmin`, `xmax`, and the minimum and maximum limits along the *y-axis* are `ymin`, `ymax`).

should have a reasonable idea for what the maximum vertical and horizontal distances are associated with ’reasonable’ choices for the initial conditions (s_0 and θ). Don’t forget the command for specifying a scale for the axis limits is `axis`. (Figure 9.4-esk maybe?)

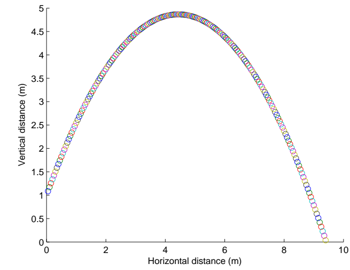

Figure 9.4: Trajectory of a ball!!

Your final task is simply to play about with the pause interval, and the model initial conditions. You can have al the trajectories appearing on the same plot if you comment out the `figure` command in your script, and open a single new figure window at the command line (» `figure`). Then each and every time you run the script, the new trajectory will be added on top. You might also try turning your script into a function so that you do not need to edit the values of s_0 and θ in the code, but pass them into the program as parameters instead (the function needs not return anything however).