7.2 *GUI Pokemon game*

Now we'll build on your excellent GUI skills and create a GUI interface for the ballistics (ball trajectory) model.

The idea of the 'game' is that you are going to launch a ball, the behaviour of which will be calculated as per your time-stepping ballistics model. Rather than simply detect whether or not the ball falls below zero (height), there will be a graphic (Pokemon) displayed and a 'hit' will be recorded if the position of the ball falls within the boundary of the graphic. The key initial conditions – initial speed and angle of the launched ball, will be set by controls in the GUI rather than set in code. Finally, there will be a series of refinements to improve the look and feel (and game-play) of the game that will introduce a few further concepts in creating good MATLAB GUIs and also new MATLAB functions. Ultimately, the GUI (app) might look something like Figure 7.7, but how the controls are positioned in the window and their relative size and shape, is pretty well much up to you. You could also control how the initial parameter values are set in a different way (e.g. using an **Edit Text** box rather than a **Slider**). Quite what buttons you want and how they are used is also a matter of personal aesthetics.

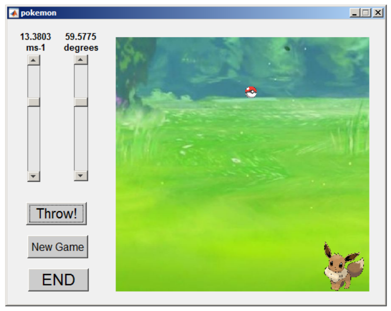

Figure 7.7: Screen-shot of he Pokemon game App.

There is quite a lot of coding to be done and the risk of a huge mess ensuing. So we'll go through this all in a number of discrete steps:

Part I Create a basic GUI interface using **MATLAB** `guide`.

Part II Load in and display the graphics needed for the game.

Part III Add in the ballistics model.

Part IV Utilizing the sliders.

Part V Create the detection (logic) needed for a successful 'catch' and associated outcomes.

Part VI Refinements to improve the look and feel of the game.

Because of the complexity of the project, the complete code (**m-file**) as well as associated `.fig` GUI file, are provided (on the course webpage). These are provided if needed for guidance (e.g. what code goes where?), only. Try your best to work through the creation of the App without this.

Example images are provided (download via the course webpage) and you can substitute your own if you prefer.

If you run into unexpected and apparently nonsensical 'issues' when you make changes and text the App, try closing the design window and any open Figure windows and type » `clear all`.

Part I – the basic GUI.

To achieve a GUI along the lines of Figure 7.7 you need to create the following objects in the window design editor (but don't create them quite yet – details will follow ...):

1. Something to display all the action and graphics in. This is pretty well much like **MATLAB** creates when you use `plot`, `scatter`, or any of the graphical functions that create a **Figure Window**. This is called an **Axes** object.
2. A **Push Button** for telling MATLAB to start calculating (and displaying) the balls' trajectory.
3. A **Push Button** for resetting the game once it is finished.[17]
4. A **Push Button** to finish the game and close the App.
5. A **Slider** (bar) to set the initial speed of the ball.
6. A **Slider** to set the initial angle of the balls' trajectory.
7. For each slider bar: a **Static text box** to display the value.
8. Also for each slider bar: a **Static text box** to display the units.

[17] This we'll only worry about making use of this in Part IV.

Make a start by running **GUIDE** at the command line. Create a new (blank) GUI. You might save it once the GUI editor window has open up[18]. **MATLAB** then opens the **Editor** and the GUI code template.

[18] **File – Save As...**

Sketch out on a piece of paper how you might lay out the objects in your GUI window before you actually start to create anything. If you have graph paper to hand, you could sketch out your design on a grid similar to the design window grid and size. Note that should should be aiming to make the **Axes** object square (i.e. the same length in both x and y dimension) as the background image we are going to use is square.[19] Also note that the **Sliders** can be horizontal rather than vertical if you prefer and if it make it easier to pack in all the objects.

[19] Later on you might want to try substituting your own background image. In this situation, you might need a different aspect ratio to the **Axes** object.

OK – to begin for real.

1. You have to start somewhere (i.e. you have to pick on one object as the first one to be created!), and the best place to start is arguably with the **Axes** object as it is the largest object in your window. Click on the **Axes** icon and drag out the position and size of the object you want.[20] By default, it is assigned a name (its **Tag** property) of **axes1**. You are not going to have so desperately many objects that it is necessarily worth re-naming it, but you can if you wish (although the text will refer to **axes1** where needed). Remember that you can move and re-size it at any point after creating it. Its position as x,y of the objects origin as well as dimensions (x-length and y-height) are indicated by **Position** at the bottom right

[20] Note that you can drag the GUI editor window larger, and you can also drag larger the gridded design area, meaning that your App window will be larger that you run the program.

of the design window. For e.g. creating an approximately square **Axes** object, you can also simply count the number of grid lines in each dimension.

Save the **.fig** file and run it[21]. You do indeed have a graph-like object with labelled axes. This is not actually that convenient (to have the axes labels when you don't need any in this particular example). In the design window – double click on the **Axes** object to bring up its list of properties. Find and edit **XTick** – delete all the tick mark numbers. Do the same for the y-axis. Close the GUI window from the previous version if it is still open, then save and re-run. Now you should see a large white square(ish) with two thin black lines delineating the axes[22], and nothing else.

[21] Note that there are two things that potentially might both need being saved – the **m-file** and the **.fig** file. If you make code changes, save the **m-file**, and if you make design change sin the GUI editor, save the **.fig** file.

[22] We could remove these black lines, but they'll get covered up later.

2. Next *Push Button* #1. Create (position and size, where- and how-ever you think best). Simplest is to leave the default name ('**pushbutton1**'). Change the text associated with the *Push Button* (property '**String**'). Label as 'Throw', 'Go', or whatever seems appropriate. Remember that you can change the default font size, family, color ... (and e.g. make bold etc.) as well as the color of the button itself (plus a host of other property options).

3. Create a 2nd *Push Button* ('**pushbutton2**') as per before. Label consistent with the GUI aim (and e.g. Figure 7.7).

4. Similarly, create 3rd *Push Button* ('**pushbutton3**').

5. Now we need a **Slider**[23] bar. These are bar with a slider ('knob') that can be slide up and down via the mouse, or moved by clicking in the bar above or below the position of the slider. By doing so (changing the position of the slider along the slider bar), you change the numerical value of the slider. We are going to use one in order to set the initial speed of the ball. So go create one (leaving the default name of '**slider1**').

[23] Not anything to do with baseball.

Because we need to link the **Slider** to our model (in terms of parameter value), we need to specify a minimum and maximum value that the **Slider** can take, as well as an initial value. These properties can be set at in the code, but we'll start off by specifying them using the design GUI tool. If you double click on the **Slider** you'll get its property list opened up. The minimum and maximum property value name are Min and Max – edit these to span a plausible initial speed range[24]. Also set a default initial value (parameter name '**Value**')[25].

[24] I used 0 to $20ms^{-1}$.

[25] I assumed $0ms^{-1}$.

6. Create a second **Slider** ('**slider2**') for setting the initial angle of the ball (*theta*).[26]

[26] Here I assumed a range of 0 to 90°, with a default of 0°.

7. Because the **Sliders** themselves do not tell you quite what value you have slide the slider to, it is a Good Idea to somewhere

display the value. We'll do this via a **Static Text** box ('**text1**') and you'll need to create one to go with each **Slider** (so you'll also have a '**text2**' named object). For now – simply leave the default text property as it is.

8. Finally, if you follow the design in Figure 7.7, you could add a further pair of **Static Text** boxes in order to display the units. This is far from essential and I'll leave it up to you whether you bother, particularly if your window is cluttered already.

That is the basic GUI design done. Save and run (having first closed any open, running, instances of your GUI program). You should have a window with all the objects discussed, but with none of them yet doing anything.

At this point it is worth quickly orientating you around the automatically-generated code **m-file**:

- At the very top:

```
function varargout = pokemon(varargin)
```

appears at the very top of the **m-file** and defines the main function. In this example, the main function is called `pokemon` (meaning the App is run by typing » `pokemon`). Remember that you do not have to edit any of this function.

- Next comes:

```
% -- Executes just before pokemon is made visible.
function pokemon_OpeningFcn(hObject, eventdata, handles,
varargin)
```

This is the function that is called just before the window is made visible and we'll edit it later in order to carry out some initial tasks (i.e. before the ballistics model itself runs).

- Then:

```
% -- Outputs from this function are returned to the command
line.
function varargout = pokemon_OutputFcn(hObject, eventdata,
handles)
```

which is mysteriously useless and we will not edit.

- The first actually useful automatically generated code is:

```
% -- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
```

This will contain the code that is executed when the 'Throw' (or 'Go') button ('**bushbutton1**') is pressed and will end up containing the complete ballistics model code.

- The function code for when second button ('**bushbutton2**') is pressed appears in order after the function associated with '**bushbutton1**':

```
% -- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)
```

We'll only make use of this towards the very end of this section is making the final refinements to the App.

- Then, the third button ('**bushbutton3**'):

```
% -- Executes on button press in pushbutton3.
function pushbutton3_Callback(hObject, eventdata, handles)
```

This will contain more more than a command to close the App (as you have programmed previously).

- The code that is called whenever the position of the first slider the appears:

```
% -- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
```

- This is then followed by a second function associated with **slider1** whose purpose is ... not obvious. Perhaps slider initialization? Regardless, we'll be ignoring the following code:

```
% -- Executes during object creation, after setting all
properties.
function slider1_CreateFcn(hObject, eventdata, handles)
```

- The final code is the pair of functions for the 2nd slider (of which we'll only edit the first function (`slider2_Callback`)):

```
% -- Executes on slider movement.
function slider2_Callback(hObject, eventdata, handles)

% -- Executes during object creation, after setting all
properties.
function slider2_CreateFcn(hObject, eventdata, handles)
```

Before we move on, you could add your fist code to the **m-file** – a close action if you click on the lower of the three **Push Buttons**. Refer to the previous sub-section and example to remind yourself how to do this. You are aiming to have the App window close when you click on **pushbutton3**, whose associated function is called `function pushbutton3_Callback`.

Save the **m-file** and re-run the App by typing its name (e.g. » `pokemon`) and the command line (first closing any already open instances of it). The App window should now close when you click on the third button. In the GUI design editor, edit the 'value' of the **String** property of this **Push Button** so that it has a logical and vaguely meaningful label.

Part II – (graphics) initialization. Note that in this section, all the code will go in `function pokemon_OpeningFcn`, after the (automatically generated) lines:

```
% Choose default command line output for pokemon
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% UIWAIT makes pokemon wait for user response (see UIRESUME)
% uiwait(handles.figure1);
```

First, we'll read in a background image ('background.jpg' – available for download from the course webpage) and then display it. We'll use the commands `imread` for reading in the graphics format (and converting it into something **MATLAB** prefers) and then `imshow` to display it. The first part is easy enough:

```
img_background = imread('background.jpg');
```

The question then becomes 'where' to display it. You might not think there is even a question in this – in the window! Except ... where in the window? We actually want the background image in the (currently) blank **Axes** area, not just anywhere in the figure window (which also have various button etc. objects positioned in it). We need to find the ID of the **Axes** object and tell MATLAB that is 'where'.[27] We can get the handle (ID) of the Axes object via:

[27] Actually, it may work without worrying about this, but we'll need to be able to specify where to position other images later anyway.

```
h_axes = findobj('Tag','axes1');
```

and then tell MATLAB that this is currently the object to put things in by:

```
axes(h_axes);
```

We then use this handle in the call to `imread`:

```
h_background = imshow(img_background,'Parent',h_axes);
```

While we're at it, we can specify the axis range for plotting the position of the ball in the Axes object, and add a `hold on` for completeness. If we also define the axis ranges (in m) as parameters (that we can use elsewhere), the complete code (so far), at the end of the automatically generated code in `function pokemon_OpeningFcn`, becomes:

```
% define grid dimensions
x_max = 10.0;
y_max = 10.0;
% read in background image
img_background = imread('background.jpg');
```

```
% set axes suitable for game
axes(h_axes);
axis([0 x_max 0 y_max]);
hold on;
% draw background
h_background = imshow(img_background,'Parent',h_axes);
```

When you run this, you should get Figure 7.8.

Next, we want a Pokemon to throw the ball at! The load-in code (which can go after the code fragment above) for the image is identical to before:

```
img_eevee = imread('Eevee.png');
```

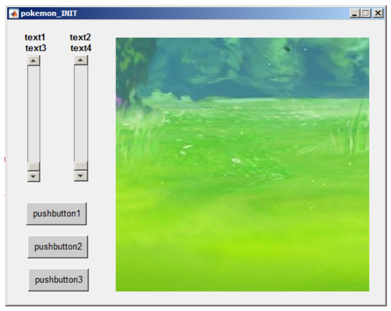

Figure 7.8: Template App with background image.

(The image itself ('Eevee.png') can be downloaded from the course webpage.) There are two complications in using `imread`, however. To see what these complications are, after the `img_eevee =` line, add the following:

```
h_eevee = imshow(img_eevee,'Parent',h_axes);
```

to also display the image. Well, it is a bit of an odd mess. By default, `imshow` tries to fit an image to the space, so that might, at least partly, help explain things.

We can start by making the Pokemon image smaller and see whether that helps us to work out what is going on. To do this, we could e.g. pick half of the size of the **Axes** object, and plot the Pokemon from the origin. A replacement line to do this would look like:

```
h_eevee = imshow(img_eevee,'Parent',h_axes,'Xdata',[0 x_max/2],...
'Ydata',[0 y_max/2]);
```

When you run this, you should get Figure 7.9.

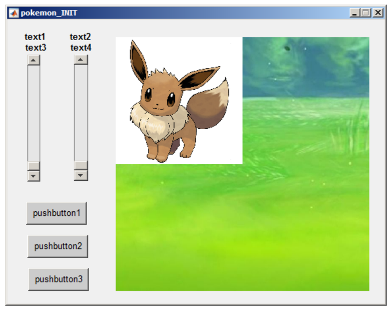

Figure 7.9: Template App with background image plus Pokemon.

You can see firstly that the Pokemon image is half the size of the space – exactly as we requested via `'Xdata',[0 x_max/2]` which says to start the image at zero on the x-axis and stretch it horizontally until half way along (`x_max/2`), and similarly for the y-axis. Except ... with `imshow`, it seems that the y-axis origin starts at the top and is positive downwards (which the Pokemon is in the top left, rather than bottom left, corner).

To cut a long story short, we can generalize the position and size of the Pokemon that is displayed (and use this at the end when we refine the App), via the following code fragment[28]:

[28] You should delete the lines starting `img_eevee =` and `h_eevee =` first. This 10-line code fragment then follows directly on from the previous 11-line one.

```
% define pokemon size
dx_pokemon = 0.2*x_max;
dy_pokemon = 0.2*y_max;
% define initial pokemon position
x_pokemon = x_max-dx_pokemon;
y_pokemon = y_max-dy_pokemon;
```

```
% read in pokemon image
img_eevee = imread('Eevee.png');
% draw pokemon
h_eevee = imshow(img_eevee,'Parent',h_axes,'Xdata',[x_pokemon...
x_pokemon+dx_pokemon],'Ydata',[y_pokemon y_pokemon+dy_pokemon]);
```

Now giving you a small Pokemon – in fact, 20% of the **Axes** size as specified in the definition of the Pokemon size parameters, `dx_pokemon` and `dy_pokemon`. If you run this, you should get Figure 7.10.

One final thing now is the background to the Pokemon image. The original format (png) is actually defined with a transparent background. **MATLAB** can make use of this with a small tweak to the code – replacing the `img_eevee =` line with:

```
[img_eevee, h_map_eevee, h_alpha_eevee] = imread('Eevee.png');
```

which grabs additional graphics information and specifically about the transparency. And after the last line (`h_eevee =`), add:

```
set(h_eevee, 'AlphaData', h_alpha_eevee);
```

which implements the transparent background and hopefully gives you Figure 7.11.

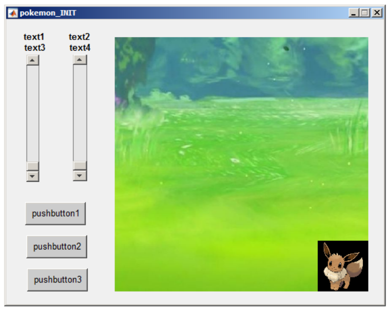

Figure 7.10: Template App with background image plus small Pokemon at bottom right.

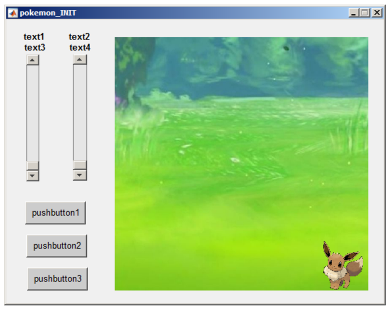

Figure 7.11: Template App with background image plus small Pokemon at bottom right, now with its transparency applied.

Part III – incorporating the ballistics model.

Here – almost all the code in this section will go into `function pushbutton1_Callback` – the function that is executed when the first **Push Button** is clicked. But before any coding – ensure that the text label associated with the first `Push Button` is appropriate for launching the ball ('Throw', 'Go!', whatever).[29]

[29] Remember – double-click on the **pushbutton1** object in the design editor and then find and edit the value of the **String** property.

Below is a simple rendition of the ballistics model. All that has been modified from a stand-alone **m-file** that would plot the trajectory of a ball, is that the creation of a figure (and associated `hold on`) is not necessary (because this has already bene done within the initialization function). either copy-paste your own version (and comment out the figure creation line), or add the below version.

```
% model constants
g = 9.81;
% model parameters
theta0 = 80.0;
s0 = 5.0;
h0 = 2.0;
% model parameters - time (s)
dt = 0.05;
t_max = 10.0;
% calculate initial velocity components
u = s0*cos(pi*theta0/180.0);
v = s0*sin(pi*theta0/180.0);
% set initial position of ball
x = 0.0;
```

```
y = h0;
% create Figure window and hold on
%Figure;
%hold on;
% run model
for t=dt:dt:t_max,
    end
    % update horizontal and vertical positions
    dx = dt*u;
    x = x + dx;
    dy = dt*v;
    y = y + dy;
    % plot current position of ball
    scatter(x,y);
    if (y < 0.0)
        break;
    end
    % update vertical velocity (horizontal velocity unchanged)
    dv = -dt*g;
    v = v + dv;
end
```

When you rn the complete App, and press the first **Push Button**, you should see the ball's trajectory plotted. Upside-down! WTF!?

Well, this does seem to be the coordinate system in this **Axes** object. We can fix this by subtracting the model calcuated height (`y`) from the maximum y-axis value (`y_max`) and adjuct the `scatter` line to:

```
    scatter(x,y_max-y);
```

Except ... we defined `y_max` in the initialization function, and its value is not available in this function, unless we define it as `global` in both, so lets do that – add the following lines:

```
global x_max;
global y_max;
```

to both

- `function pokemon_OpeningFcn`
- `function pushbutton1_Callback`

(before any of your other code in these files, but below anything that **MATLAB** generated automatically in the first place).

It works, and in the right direction (for 'up'), but it is hardly **iTunes** grade App material. What we can do, is to replace the point plotted by `scatter`, with an image.

At the top of `function pokemon_OpeningFcn` (after the `global` declarations) load in a ball image:

```
[img_ball, h_map_ball, h_alpha_ball] = imread('Pokeball.png');
```

(using the full format of returned parameters because we'll make use of its transparency). We'll then define the size of the ball:

```
dx_ball = 0.05*x_max;
dy_ball = 0.05*y_max;
```

and finally, in place of `scatter ...`, write:

```
    h_ball = imshow(img_ball,'Parent',h_axes,'Xdata',...
    [x x+dx_ball],'Ydata',[y_max-y y_max-y+dy_ball]);
    set(h_ball, 'AlphaData', h_alpha_ball);
```

The first of these final two lines, displays the image given by the parameter (ID) `img_ball`. It ensures it is displayed in the axes area pointed to by `h_axes` (and because of this, you also have to define `x_axes` as global[30], i.e. `global h_axes;`). Its size is `dx_ball` by `dy_ball`. Its x-coordinate is simply `x` (hence the image goes from `x` to `x+dx_ball`) and its y-axis coordinate ... well, don' worry about it, after much trial-and-error, it works. Now you should have something like Figure 7.12 when you run it.

[30] Directly underneath the other two `global` definition lines AND in a similar position in the initialization function: `function pushbutton1_Callback` .

To finish this section off, we'll improve how the trajectory f the ball is displayed. Firstly, we could add a delay between each addition of the ball image, rather than them all sort of appear at once. After the `set ...` line, add:

```
    pause(0.005);
```

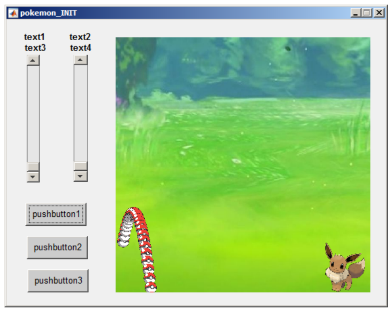

Figure 7.12: App with ball trajectory trail.

This is some improvement visually. We could also remove the previous ball image, so that only one ball image is displayed on the screen at any one time, hopefully giving the impression of movement. Since we were good and obtained the handle (h_ball) of the ball image when we displayed it, this gives us a means to tell **MATLAB** to get rid of it again. Now, after the `pause` line, add:

```
    delete(h_ball);
```

which simply deletes the last ball image object that was plotted.

Now when you run it you should see a single ball image that follows the trajectory that you calculated with your time-stepping ballistics model.

Part IV – utilizing the sliders.

So far it is not much of a game – the values of the parameters determining the initial speed and angle of the ball are set in the code. You could always edit the code, save, and re-run to replay the game with a different throw, but ... really(?)

The **Sliders** are there to allow you to adjust the two key parameter values and the 'Throw' (/'Go') button can be re-clicked on to then re-run the game. The **Sliders** are set up such that when you move the slider, its value changes. In designing the GUI and creating the objects you have already set the min and max values of the **Sliders** to something reasonable. What remains is to obtain the value of each **Slider** and pass that to your ballistics model.

The first step is to read the new **Slider** value when the slider is moved. Taking the example of the first **Slider** ('**slider1**') which controls the initial speed of the ball – we first need to request the handle (ID) of this **Slider**. As before, we use the `findobj` function:

```
h = findobj('Tag','slider1');
```

which simply asks for the handle (passed to variable `h`) of the object whose '**Tag**' is '**slider1**'. You then[31] use the `get` function to get the '**value**' (one of the properties of the object):

[31] On the next line.

```
s0 = get(h,'Value');
```

where here the value is assigned to the variable `s0` (initial speed). These two lines of code go in `function slider1_Callback` just after the comment lines (there is actually no other code (automatically generated) in this function as it currently stands).

While we're here editing this function, what else might be helpful to happen when the slider is moved and its value changes? Although from creating the **Slider** object you know (unless you have forgotten) what the min and max **Slider** values are, you would still be somewhat guessing what its exact (or even rough) value was. During the GUI design phase, you created a pair of **Static text** boxes for each **Slider**. One of each pair was intended to display the **Slider** value. So lets do this now. The **Static text** box for the value display was called (its **Tag**) **text1**[32].

[32] At least, it was in my GUI design – check the name of yours.

Once again, before we can change any of the properties, we need to determine the handle of the object. For **Static text** box **text1**, the code would be:

```
h = findobj('Tag','text1');
```

(this should be starting to become familiar to you by now ...).

To set its value, which in this case is a text string, we write:

```
set(h,'String',num2str(s0));
```

where `num2str(s0)` converts a numeric value into a string (as you have seen before). These two lines of code will go after the first two in the same function (as you need to have obtained the value of so before you can use it to change then text box display).

At this point you may as well save and re-run. Now, when you drag and release the slider for initial speed, its new value is displayed above it in the text box. At least, this should be what happens ...

Write the analogous four lines of code for the other Slider, which will go in `function slider2_Callback`. Now the parameter value being read and displayed in the text box is the initial angle of launch, `theta0` (of whatever you prefer to call the parameter).

Again – save and test what you have so far. This should now be two **Sliders** that are linked to two **Static text** boxes such that when the slider is moved, the new values are displayed.

There is one final step to take. If you change either or both **Slider** values and click on 'Throw' /'Go', the trajectory of the ball is the same as before – you are not actually changing the parameter values used to initialize the ballistics model yet. Recall that variables within *functions* are *private* – they cannot be 'seen' outside of the function their value is set in. Unless you declare them as `global` variables.

So, in each **Slider** function, you need to declare the respective parameter (`s0` or `theta0`) as `global`. This will need to be the first line of the code (after the comment lines and before the four lines of code you inserted). You will also need to add the global declarations at the start of the pushbutton1 code where your model lives (`function pushbutton1_Callback(hObject, eventdata, handles)`):

```
global s0;
global theta0;
```

You then need to comment out the lines that set your initial model parameter values:

```
%theta0 = 80.0;
%s0 = 5.0;
```

You can test it now, and if you do, you might find that nothing appears to happen if you press 'Throw'. Only if you change the slider positions does anything (i.e. a moving ball) happen. We have created the situation where the ballistics model takes it values for initial speed and angle from the parameters `s0` and `theta0`. The only place in the code in which these values are set are the **Slider** functions. BUT, the **Slider** functions are only called when the slider is moved. So on starting the App, unless you first move the **Sliders**, the values of `s0` and `theta0` are undefined[33].

[33] Invariably, undefined variables in code are assigned a value of zero, but you should never try and use a variable whose value has not somewhere been defined.

What to do? Well, recall there is the function that is called when the App first starts up and in which we loaded up various images etc. In this function, we could also check the value of each **Slider** (even though the slider could not have been moved yet), set the parameter values, and display the **Slider** values in the **Static text** boxes.

At the end of the code in `function pokemon_OpeningFcn`, add:

```
% read in default model parameters and set labels
h = findobj('Tag','slider1');
s0 = get(h,'Value');
h = findobj('Tag','text1');
set(h,'String',[num2str(s0)]);
```

```
h = findobj('Tag','slider2');
theta0 = get(h,'Value');
h = findobj('Tag','text2');
set(h,'String',[num2str(theta0)]);
```

which is pretty well much just an amalgamation of the code you have added to the two **Slider** callback function. The last final piece is to remember that the initial Slider values you read and set `s0` and `theta0` on the basis of, cannot be seen outside of this function. So at the top, along with the other `global` statements, make `s0` and `theta0` global to.

Note that if you do not like the new defaults for `s0` and `theta0`, you can always edit the properties of the **Sliders** in the GUI design editor window thing.[34]

[34] Equally, you could have coded in defaults and then set the **Slider** values to be these defaults when the App starts up. The process is basically exactly the same as for setting the **Static text** box string values.

Part V – pokeball/Pokemon collision detection.

Part VI – final game refinements.