Online Resources Documentation

Chemical Structure and Reactivity

Jamie Wiles

September 1, 2016

Contents

1 Introduction

This documentation is designed to help someone to understand what is going on in the code for the web-links, in case something goes wrong in the future or in case anything needs to be changed. It is not intended as a comprehensive guide on how to use JSmol, JSXgraph and jQuery: for this, one must refer to their respective documentations. This document simply tries to explain why I have done certain things in certain ways, so that whichever poor soul has to deal with my code in the future can see some of the method behind the madness[1]. (And so that they don't fall into the same traps as I did!)

The web-links for this book fall into three main categories: 3-D models of molecules, iso-surface plots, and interactive graphs. Web-links 2.2 and 14.1 are combinations of these, and web-link 2.1 is rather different to all the others, essentially being an interactive image slideshow (see §4 for details). The molecular models and isosurfaces are both created using the JSmol JavaScript library and HTML5, whereas the interactive graphs were created using the JSXgraph library. Details on how both these types of web-link were implemented can be found in §3 & §2 respectively.

2 JSXgraph resources

2.1 Background

JSXgraph is a JavaScript library for producing interactive graphs on a web-page. It is a relatively new library, meaning the documentation is a bit sparse and finding online support when trying to do something a little more complicated can be difficult. However, it does exactly what we need it to for all of these resources and hopefully the documentation will continue to improve as the library matures.

[1] Also, I am aiming this document towards someone with a similar computing ability as I had when I started developing these resources: a NatSci student (Part II Physics) with some programming experience but limited experience in web development and JavaScript. If you're reading this as a more seasoned programmer, I can only apologise and hope I haven't said or done anything too stupid...

2.2 The fundamentals

2.2.1 Basic set-up

To include JSXgraph on a webpage, the documentation says you need to use 2 files. The first is the `jsxgraphcore.js` JavaScript source file and the other is the `jsxgraph.css` CSS file. However, in these web-links I did not link to the CSS file - this is because doing so produces a blue box around the graph which doesn't look that great. It seems though that the `orc.css` file sorts out the fonts etc. in the JSXgraph app so everything still works fine without `jsxgraph.css`.

To make the JSXgraph app, you *need* three things:

1. The link to the JavaScript source,

   ```
   <script>jsxgraphcore.js</script>
   ```

2. Some JavaScript, including at least one JSXgraph board, *viz.*

   ```
   var board = JXG.JSXGraph.initBoard("GraphName", board_attributes);
   ```

3. A reference to the JSXgraph in the HTML, including the same name as you used when making the board in the first place:

   ```
   <div id="GraphName" class="jxgbox" style="width:600px; height:600px;"></div>
   ```

Note: The JavaScript for the JSXgraph must come after the JSXgraph `<div>`, otherwise it won't work. So the script cannot be put into the header, as is the case for all of the JSmol based web-links.

You may have noticed the mysterious variable, `board_attributes`, in the code for making the JSXgraph board. What is this? It's basically just an object, which contains all of the properties of the board itself (c.f. the `Info` object used in the JSmol, §3.2). For full details of all the attributes you can choose from, refer to the **JSXgraph documentation**. An example of a typical set of attributes used in these weblinks is given below:

```
var board_attributes = {
        boundingbox: [-12, 22, 12, -3], //two coordinates: [upper-left, lower-right]
        axis:false,          //don't show the default axes - they don't look very nice
        grid:false,          //also don't show the grid
        showCopyright:false, //don't show JSXgraph logos etc.
        showNavigation:false,//don't show the navbar - we don't ever need it
};
```

These attributes could of course just be typed directly into the `JXG.JSXGraph.initBoard` function, but I found that saving them as variables looks neater and makes it easier to edit. Plus, when multiple objects share the same/similar properties (e.g. the x and y axis), it is easier to refer to the same object rather than typing out the same attributes twice.

2.2.2 Producing lines, points, graphs etc

Producing lines, graphs etc. is pretty straightforward, though there are some subtleties worth noting.

For example, to produce a line the syntax is:

```
var line = board.create("line", [[x1, y1],[x2,y2]], line_attributes)
```

where `line_attributes` is an object containing all the basic properties of the line (c.f. `board_attributes`). These properties include the width of the line, its colour, whether it should end in an arrow etc... (look at the documentation!). Note that I've used `board.create`, because I gave the board the rather inventive variable name, board. If you called your board something more exciting, like `brd`, you'd have to use `brd.create` instead.

Points, and text labels can be produced in a similar way: check out the documentation. Functions are slightly more tricky though, so I'll include an example below. This plots a graph of $f(x) = sinc(x)$ between $-\pi/2$ and $\pi/2$. Note that the function you are plotting, $f(x)$, needs to be included as a function of x (this is reasonably intuitive!)

```
var pi = Math.pi;
var graph = board.create("functiongraph", [function(x){return Math.sin(x)/x}, -pi/2,
 ↪  pi/2], attributes)
```

2.3 Interactive elements

2.3.1 Sliders

All of the JSXgraph weblinks are interactive in some way, and most of them use a slider. These sliders often the positions of lines, the nature of functions and the region in which they are plotted. A slider is produced using

```
var slider = board.create("slider", [[starting coordinate of the slider],[ending
 ↪  coordinate],[minimum value, starting value, maximum value]])
```

and its value can be returned using `slider.Value()`.

Important: if you want the x coordinate of a point to take the value of a slider (say), simply using `slider.Value()` as the x coorinate will not work. **Whenever you want a something to change with the slider in real time, you MUST include it as `function(){return slider.Value()}` or similar.**

2.3.2 Radio groups, check-boxes and menus

Radio groups and drop-down menus aren't included natively in JSXgraph. However, this can be worked around by creating a 'text' object, and then inserting the HTML for a check-box/radiogroup in the place of the text, i.e.

```
board.create("text", [0, 1, "Hello"])
```

puts the word 'Hello' on the board at (0, 1) whilst

```
board.create("text", [0, 1, '<form action="">
  <input type="radio" name="greet" value="hello"> Hello
```

```
  <input type="radio" name=""greet" value="goodbye"> Goodbye
</form>'])
```

produces 'Hello' and 'Goodbye' radio buttons at (0, 1). Note that the entire HTML is included as a string: this means that you should either need to use '' around the outside of the HTML and "" on the inside (say) to make this work properly.

Alternatively, you can just place the interactive elements outside of the HTML. However, you get better control over the visuals of the app if you place them on the inside.

2.4 Web-links 18.1-18.3: Quantum Mechanical Potentials

Here I describe some of the online resources in more detail, and any particular difficulties I encountered and how I got around them.

2.4.1 `for` loops

These three web-links plot the wavefunctions and energy levels for different types of QM potential: the infinite square well, the quantum harmonic oscillator and the Morse potential. The nature of the wells can be altered with sliders, e.g. the width of the square well, the depth of the Morse potential or the value of k/m for the QHO.

In three all web-links, the wavefunctions and energy levels are all functions of n (or v), and so it makes sense to plot both of these using `for` loops. However, I had *major* problems getting these to work dynamically with the sliders. For example, if you try plotting the each energy level `for` loop that counts up to 15, then to begin with, everything works fine. However, once you change something with the slider, **all of the levels default to** $n = 15$, which is rather a problem.

In the end, I found that the solution to the problem goes as follows:

1. Make a function that takes one parameter, n
2. In this function, plot the nth energy level and/or the nth wavefunction
3. Make a `for` loop that calls this function with different values of n.

```
var make_level = function(n) {
    board.create("line", stuff);
}

for(n=0; n<15; n++) {
    make_level(n);
}
```

For some reason, this is now perfectly well behaved when you change things with the sliders.

2.4.2 Classically allowed/forbidden regions

In web-links 18.2 and 18.3 the wavefunction is coloured blue or red depending on whether it is in a classically allowed or forbidden region. Unfortunately there is no way of giving a region of a graph a different color in JSXgraph so instead 3 separate plots had to be produced: the classically allowed region of the wavefunction and the inner and outer tails. For the Morse oscillator (18.4), there are some `if/else` statements in the bounds of the graph: this is because in some cases the classically allowed region may extend out of the graphing region making it look a bit crap.

2.4.3 Radio buttons

These three apps use radio buttons to switch between wavefunctions, squared wavefunctions and energy levels only. The radio buttons are included in the HTML, not the JSXgraph board itself: this is because when I made these I was not aware of the trick described in §2.3.2.

The radio buttons call the function `changePlots` which takes one parameter to tell it which option is selected. The function deletes all of the wavefunctions/squared wavefunctions and then replots them in the desired form. The wavefunctions / squared wavefunctions are saved into arrays: this is because they can then be all be deleted at once using `board.removeOject(array)` rather than having to delete them one by one.

2.5 Web-link 12.1: Reaction Profiles

2.5.1 Steady state/pre-eqm solutions

In this web-link the analytic solutions for the A$\longleftrightarrow$B$\longrightarrow$C reaction are shown, and the solutions given by the steady-state and pre-eqm approximations can be toggled on or off using checkboxes. These checkboxes both call a function which increments a counter, i or j. Depending on whether i is odd or even, the function will either delete the steady-state graphs or create them. Both checkboxes are independent and so the user can choose to display both approximations, only one or neither.

In this app, the solutions are all taken to be of the form

$$c_i(t) = a_{ij}e^{-\lambda_j t}$$

where the a_{ij} and λ_j are constants. Exact expressions for these can be found using the initial conditions that $c_1(0) = 1$, $c_{2,3}(0) = 0$ and conservation of mass. Using the steady state or pre-eqm approximations leads to simplified expressions for the a_{ij} and λ_j. However, **in this app only the simplified a_{ij}: the original exact expressions for λ_j are kept.** This is because the old Java applet did it this way, and using the approximations for the λ_j gives some really messy results in certain limits.

2.5.2 Axis scaling

In this web-link the user can change the horizontal scale using a slider. This slider doesn't actually change the x-range, as this would also move the sliders. Instead, it introduces a scale factor into the graph, $f(x) \rightarrow f(2x)$ say, that squashes/stretches the graph. The x-ticks and their labels are produced manually and are shifted along with the slider to give the illusion of the x scale being changed.

2.5.3 Energy level cartoon

This cartoon is very schematic. It illustrates how the activation energy for a step in the reaction goes as the log of the rate constant. I fudged in constants and offsets in order to make the cartoon look 'right'.

The curves between levels are made using a curve fit. JSXgraph has many types of curve fit, though I found that some looked a little misleading as they tried to smooth things out a bit too much. I opted for the `JXG.Math.Numerics.bezier` curve fit. This takes in an array of JSXgraph points (i.e. `p[0] = board.create("point", ...); p[1] = board.create("point", ...); ...`) and will then draw a curve fit going **through** every **third** point (`p[0], p[3], ...`) with intermediate points *shaping* the curve. I placed these two intermediates right above the first point and right next to the third: this made the curve look about right.

2.6 Web-links 4.2 & 4.3: 2D Density plots

Both these web-links contain MO diagrams, 1D plots of the wavefunctions and also 2D density plots. These density plots were a bit of a nightmare, but I used a few workarounds to get a balance between performance and functionality.

The density plots are produced by producing an array of squares. These squares are made using the function `make_2D_MOs(nx, ny, j)`, which has three arguments:

1. `nx`, the number of places from the centre in the x direction (integer, positive or negative);
2. `ny`, the number of places from the centre in the y direction (integer, positive or negative);
3. `j`, the index of the square to store into an array (positive integer)

There are also some variables `a` and `b` that are used to scale the whole plot by changing the size/spacing of the squares.

To make these squares look like a density plot, their opacity is modulated according to the value of the wavefunction: these are produced by the various `opacity_functions`, which are functions of `nx` and `ny`. This is achieved by setting the attributes `opacity` & `highlightOpacity` in the functions themselves to the `opacity_functions`, *viz.*

```
1 board.create("polygon", coords, {opacity:function(){return
  ↪ Math.abs(opacity_functions1(nx, ny)), ...})
```

Note that these functions can be positive or negative so the mod must be taken before using as an opacity. However, the sign can be used to determine whether to color the square red (for positive) or blue (for negative):

```
fillColor:function(){if(opacity_functions2(nx, ny) < 0){return 'blue'}else{return
↪  'red'}}
```

2.6.1 Improving performance by using 2 boards

It turns out that loading hundred/thousands of squares an dynamically changing their opacity makes things very slow. In fact, even just having thousands of squares on the board with a static opacity still makes the rest of the board laggy. I got around this by making two boards, side by side: the fist with all the sliders and 1D graphs and the second for the density plots. This was quite easy: one just needs to make two `divisions` within the same `<td></td>`, and ensure they have the same height in pixels. You can make and edit two different boards within the same `<script>` tags as long as you give them two different variable names, e.g. `board1` & `board2`.

We want our plots on board 2 to update when the slider is changed on board 1. You can actually make one board the child of another, meaning that it will be dependant on it and change whenever you interact with its parent. However, this updates the density plots much too often and so things still ran rather slow. Instead, I set it so that the density plots **only update when the slider is released**, using the event

```
bond_length_slider.on('up', function(){board2.update()})
```

2.6.2 Changing the plot type

The type of plot can be changed using the radio buttons. I experimented with a few ways of changing the plots, and it seems that the fastest way is to change the function going in to the opacity and then updating the board. This is done by setting the opacity to the variable `opacity1`, and setting this variable to `opacity_functions1`, `opacity_functions3`, or `opacity_functions5` accordingly.

Originally I used a much slower method and so I including a 'Loading...' division in the HTML. This is displayed whilst the density plot is being changed. I have left this is because reloading can still take about 1s in Firefox. This was achieved using JavaScript's `setTimeout` function and some simple jQuery to show/hide the necessary divisions.

2.7 Web-links 7.1 & 7.2: Boltzmann and Gibbs

These two are relatively simple, though they both feature a 'speedometer' which I should probably explain.

In 7.1, the speedometer indicates the value of the energy level spacing divided by k_BT. It therefore changes when you move the either the temperature slider or the energy level spacing slider.

The way that this works is as follows. The outside of the speedo is constructed using a semicircle (`board.create("semicircle", [left_corner, right_corner])`) and the bottom by making a line. The speedo arrow is then simply another, from the centre of the semicircle to some point on the outside.

The position of the arrow is determined by the angle it makes from the x axis, ϕ: its end then has the coordinates $(r\cos\phi, r\sin\phi)$ where r is the semicircle radius (taking the centre to be at the origin). The angle ϕ depends on Δ/k_BT, and should depend logarithmically so that the speedo dial moves smoothly. With minimum and maximum values of 0.01 and 100 and a middle value of 1 on the speedo, a suitable function to convert from Δ/k_BT to ϕ is

$$\phi(\Delta/k_BT) = \pi/2 - \pi/4\log(\Delta/k_BT)$$

In the script, this is implemented using the function `ratio_to_rads`, which is then called when making the dial. Similar functions are used in 7.2, though the numbers are different (and are fudged a bit to get it to look right!)

2.8 Web-link 2.3 Hydrogen-like wavefunctions

This web-link allows you to plot up to 3 different hydrogen wavefunctions (radial part only) or RDFs. There are a few interesting things to point out on this web-link...

2.8.1 Multiple boards

2 boards are used, one for the graph (`board1`) and the other for the UI elements (`board2`). This is so that the user can change the scale of the graph without the UI elements moving around. The script `board2.addChild(board1);` makes the board with the graphs the child of the UI board: this means that whenever something is changed in the UI panel, the graph panel updates immediately.

2.8.2 Changing the scales

Like web-link 12.1, you can change the x and y scales in this graph using sliders. However, in this case the actual scaling of the board changes, rather than applying transformations to the graphs. This is achieved using

```
1    y_slider.on('drag', function(){board1.setBoundingBox([ -x_slider.Value()/10,
↪   y_scaling(), x_slider.Value()+x_slider.Value()/10, -y_scaling()/6]); });
2        x_slider.on('drag', function(){board1.setBoundingBox([ -x_slider.Value()/10,
↪   y_scaling(), x_slider.Value()+x_slider.Value()/10, -y_scaling()/6]); });
```

The 4 numbers in the `setBoundingBox` again determine the top left and bottom right coordinates. I've set this up so that the axes don't move - I thought this would be distracting. The x ticks and y are also dependant on these sliders so that they stay the same size on the webpage - i.e. when the y axis is stretched, the x ticks get squashed so they look like they're the same size.

2.8.3 Dropdown menus

I made all the dropdown menus using `for` loops, to make things concise. However, some things weren't 100% compatible with `for` loops and so the code isn't as understandable as I might like. This is the reason for all the `if` and `else`'s in the `changeN` function.

Basically, there are two functions: `changeN` called when an n dropdown changed and `updatePlot` called when an l dropdown changed. Both take in a value(e.g. "1s", "2s"...) and another value, $n = 0, 1, 2$ which tells the function which of the three plots is being changed. `changeN` plots the $l = 0$ orbital for the value of n chosen and also deletes the l dropdown, remaking is with the new values of $l = 0, ..., n - 1$. `updatePlot` simply updates the plot, but does so using

```
plots[n].Y = function(x){return plot_1s(2*Zeff[n]()*x/1, Zeff[n]())};
```

I did it this way rather than deleting/creating new plots because I wanted plots that weren't being displayed to also be updated, so that once their checkbox was ticked the correct thing would be on view.

2.8.4 Toggling plots

The plots are toggled using checkboxes and counters, as in the other cases. However, instead of these deleting the plots or recreating them, they instead toggle the visibility attribute between being false and true:

```
plots[0].setAttribute({visible: true});
```

2.8.5 Radio Buttons

The radio buttons switch between plotting radial parts of wavefunctions and the RDF. This is done by changing the function that all of the graphs refer to: the functions `plot_1s`, `plot_2s` etc. Copies of these functions are saved as the variables `orbital_1s`. The `plot_1s` functions can then be redefined in terms of the `orbital_1s` functions to toggle between the wavefunctions and RDFs: this is what the function `changeRadialRDF` is doing.

2.9 Online Resources

A number of online resources may be useful:

1. **The JSXgraph wiki**. This gives good explanations on how to implement all of the basic elements in JSXgraph but doesn't go into many of the details beyond that. The examples are reasonably useful though unfortuantely they aren't very well commented so deciphering them can be a bit of a task.
2. **The JSXgraph documentation**. This is the place to go when you want to find out everything you can do with a JSXgraph object, say a line or a polygon. However, whilst it gives lots of

information it doesn't really go much beyond the specifics, so it isn't all that helpful as a starting point. Definitely start with the wiki pages and use this as a supplement when needed.

3. The **JSXgraph Google Group** forum and the JSXgraph pages on Stack Overflow are quite useful when you have a specific problem that you need to solve.

3 JSmol resources

3.1 Background

All of the weblinks containing interactive 3-D models of molecules, or iso-surface representations of atomic/molecular orbitals, were created using JSmol/HTML5. The web-links were previously created using Jmol, displaying the molecules/orbitals in a Java applet. However, using a Java applet comes with a number of problems...

1. Google Chrome no longer supports Java, and so Chrome cannot run Jmol applets
2. In order to run the applet on Firefox or IE, one must first add the site domain to the exceptions list within the 'Configure Java' application. This probably isn't worth the effort.
3. Even once the site is put onto the 'allowed' list, a security warning pop-up will still appear every time you load a new page containing a Jmol app

...and many, many more.

In response to the diminishing support for Java, the makers of Jmol made JSmol, an update to Jmol that can be run using JavaScript and HTML5 only. Fortunately, they made JSmol such that Jmol applets could be converted to JSmol with relatively little hassle, and just a few systematic changes in syntax. The fundamentals of the JSmol syntax are given in §3.2. However, some of the resources decided that they would rather not be converted into JSmol, and so were a bit of a pain in the arse. This was due to there being some additional changes in syntax required, (e.g. with the iso-surfaces), for rather unclear reasons. These annoying details are given later in this section, so hopefully they won't cause too much of a headache in the future.

3.2 The fundamentals

In order to include JSmol in a page, you *must* link to two files. The first is `JSmol.min.js`, which can be downloaded from the JSmol website. This JavaScript source should be included in the usual way: by calling `<script src="JSmol.min.js"></script>` in the document header. The other file that must be included is actually a folder, `j2s`, that is used to covert from Java to JavaScript. This should actually be referenced in the main JavaScipt code when JSmol 'applet' is declared (more on this later).

Including a JSmol 'applet' on a webpage can be done quite elegantly by using jQuery-like syntax[2]. In the document `<head>`er, you first include the usual `$(document).ready(function(){})`, where

[2]Note that you don't actually have to include the jQuery source file to do it like this.

the curly braces will contain all of the JSmol functions. This ensures that the webpage doesn't try to do anything fancy until the bare bones of the HTML have loaded.

In most of the web-links, I chose to use two selectors: "#main" and "#interactivity". As their names suggest, the "#main" contains the main JSmol application (or, to put it properly, the Jmol JavaScript object) and the "#interactivity" contains the radio buttons, check-boxes, drop-down menus etc. I chose to do it this way as it makes producing the desired page layout relatively straightforward, though in some cases it was easier to separate the interactivity into smaller divisions e.g. "#menus" & "#radios".

An example from weblink 1.1 is included below:

```
1  $(document).ready(function() {
2          Info = {
3                  width: 400,
4                  height: 400,
5                  debug: false,
6                  j2sPath: "../jsmol/j2s",
7                  color: "white",
8                  disableJ2SLoadMonitor: true, // makes loading look neater
9                  disableInitialConsole: true, // makes loading look neater
10                 use: "HTML5",
11                 readyFunction: null,
12                 script: "frank on; zap; load \"B2H6.ent\"; background white; color BONDS
   ↪  grey; wireframe 15; select boron; color yellow; select atomno=2,atomno=6; color
   ↪  pink;select all; spacefill 15%;connect (all) delete; moveto 0.5 -204 977 70 97.6
   ↪  131"
13         }
14
15         var RadioGroup1 = [
16                 ["select all; spacefill 15%","25%","checked"],
17                 ["select all; spacefill 40%","50%"],
18                 ["select all; spacefill 60%","100%"]
19         ];
20
21         $("#main").html(Jmol.getAppletHtml("JSmolApplet",Info))
22
23         $("#interactivity").html(Jmol.jmolBr()
24                 +Jmol.jmolHtml("<p>  Space filling  ")
25                 +Jmol.jmolRadioGroup(JSmolApplet, RadioGroup1)
26                 +Jmol.jmolHtml("</p><br><p>  Show B&#8212;H close
   ↪  contacts  ")
27                 +Jmol.jmolCheckbox(JSmolApplet, "connect (all) delete; connect 0.1 1.5
   ↪  (boron) (hydrogen) modifyorcreate; color bonds grey; wireframe 15","connect (all)
   ↪  delete","",false)
28                 +Jmol.jmolHtml("</p>")
29         )
30 });
```

The `info` object on line 2 sets up the main JSmol app. Note the attribute `j2sPath`: this links to the

j2s folder previously mentioned. The `use: true` attribute ensures that JavaScript is used to produce the app, rather than Java.

Line 11 makes the JSmol app. The name allocated to it, `MUST be referred whenever a radiogroup` menu or checkbox is made that will interact with the JSmol app. The syntax for making these interactive elements can be seen in lines 13 onwards.

Note that the `script` attribute uses Jmol script: full details on this can be found in the Jmol documentation. Using Jmol script in JSmol is basically the same as in Jmol, though there are some small differences when making the isosurfaces.

3.3 Iso-surfaces

3.4 Online Resources

4 Atomic Orbitals as Cubes

4.1 Basics

I decided to create this app using jQuery and jQuery UI, as the UI allows you to create nice sliders, buttons and dropdowns and jQuery lets you easily change a division on a webpage. This app is basically just a slideshow of images - nothing is being calculated here. The slider changes the image being displayed, and the dropdown menu changes the type of orbital being viewed by changing the type of filename being loaded.

The images of the orbitals are saved as `.jpg`s. There are 32 images for each orbital, and the images are stored in the form 1s(0).jpg, 1s(1).jpg, 1s(2).jpg, ...; 2s(0).jpg, 2s(1).jpg etc. The images are displayed in a `<div>` with the selector `theImage`. The slider is produced by making a `<div>`, and then using the jQuery syntax

```
$('#slider').slider({attributes})
```

These attributes include the maximum and minimum values of the slider, and also the `slide` attribute that tells the webpage what to do when the slider is moved. In this case, the slide attribute is

```
1 slide: function(event, slider) {
2         index = 31 - slider.value;
3         $('#theImage').attr('src', imgstring + "(" + index + ').jpg');
4 }
```

where `imgstring` can take the strings `'1s'`, `'2s'` etc., and is determined by the dropdown menu. Note that I've taken the index not as the slider value but as its maximum value, 31, minus the slider value: this is so that the first image is displayed when the slider is at the top (for some reason it wont let you put the 0 at the top of the slider very easily).

4.2 Animation

To get this thing to animate, you just need to get it to change the image sequentially every few milliseconds until all the images have been displayed. This is actually trickier than it sounds. In JavaScript there is a function, `setTimeout(function(), time)` that will execute the function after the predefined amount of time has passed. However, simply repeating a `setTimeout` function within a for loop did not have the desired effect - I think because JavaScript is single threaded.

In the end, the animation was done using a recursive function- a function that calls itself. Specifically, in this function `animateCube` there is a `setTimeout` function, within which `animateCube` is called again while the counter `i` is less than 32. This all looks quite complicated, but what it does is basically build up timeouts to execute the script at fixed intervals. This seems to be the simplest way of achieving this effect from what I've read on Stack Overflow.

4.3 Online Resources

1. **The Codecademy jQuery course**. Whilst this course isn't the best, I found it quite a useful tool for picking up the basics of jQuery and jQuery UI so that I could produce this weblink. I went into the course with a good background knowledge of JavaScript, HTML and some basic knowledge of CSS.
2. **The jQuery documentation**, though I didn't really use this
3. **The jQuery UI documentation**. This was very useful in learning how the UI elements worked and, in particular, their attributes and how to return their own values. The examples are particularly useful.
4. Stack overflow was also very helpful, particularly in getting to grips with the `setTimeout()` function and producing the `animateCube` function.