**Imperial Visualisations**

****

*Functionality Guide*

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Last updated** | **Updated By** | **Reason for Update** |
| 1.0 | 20.06.2019 | Cyd Cowley | Created initial skeleton |

**Introduction**

Welcome to the Imperial Visualisations developer team! It’s expected that a new developer has no previous experience in any web-based language; all we expect is knowledge of code and the enthusiasm to learn. This document will take you through the basic building blocks of an imperial visualization, as well as recommendations on how to write clear, effective, and optimised code.

**HTML**

HTML is the backbone of any webpage, and this is definitely the case for Imperial Visualisations. HTML is less of a language and more of a collection of blocks or ‘elements’ that define what goes on your page. These elements are defined using opening and closing tags. For example, if I wanted to have an element displaying “hello world”, I would have an open <div> tag, followed by the text, followed by a closing </div> tag. So overall the element looks like:

<div>hello world</div>

These elements can contain anything from text to a slider to a plot, and can be styled to your preference! This styling – from size of your element to the text colour – can all be specified using another language: CSS. For more on CSS and styling HTML, check out the Style Guidelines document by Imperial Visualisations!

When constructing your own HTML, it is likely to be comprised of two sections: a head and a body. In the head, all of your external packages apart from your JavaScript file(s). For example, if I was using mathjax to write my equations, I would import mathjax in the head. The body contains the actual elements that appear on the page. Then, at the end of the body you should import your JavaScript files. This is done because … .

For more tutorials on HTML, here are a few useful resources:

* W3 Schools: <https://www.w3schools.com/html/html_intro.asp>
* CodeAcademy: <https://www.codecademy.com/learn/learn-html>

**JavaScript**

HTML allows us to define everything on our page, but cannot actually **do** much. For the most part, HTML is static, and cannot perform complex equations or plotting. This is where JavaScript comes in. Imperial Visualisations uses JS (JavaScript) to perform physics and mathematics calculations, plot, and perform interactivity. For the most part, JS is structurally similar to python; it has functions that take input variables equal to a string, float, integer, or Boolean. However, if you are coming from a Python background, there are a few important differences to take note of:

* Defining variables for the first time needs to be preceded with either **Let**, **Var**, or **Const**. For example, the start of your JS code could say const g = 9.8 . The difference between these variable declarations is:
  + **Let** exists in a block-scoped namespace. Ie, if it is defined in a function then it stays in that function.
  + **Var** is very wishy-washy. Do not use it.
  + **Const** is also block-scoped, but cannot be reassigned.
* Rather than using indentation to define where a for loop / function is, JS uses curly brackets. For example:
  + This function returns the square of the input value:

function square(x) {

return x\*x;

};

* + This for loop counts from 1 to 10:

for (let i=0; i<10; i++) {

console.log(i);

};

* + This if statement prints ‘yay’ if happy is true and ‘aww’ if happy is false:

if(happy==true){

console.log(“yay”)

}else{

console.log(“aww”)

};

**JQuery**

JQuery is a plugin to JavaScript to allow easier interactivity with HTML elements. After importing JQuery in your HTML head, you can then treat any html element as a pseudo JS variable. To do this you can reference an element by id:

let Length = $(“#lengthslider”);

where the id is “lengthslider” in this case. Or by class:

let Length = $(“.slider”);

where the class is “slider” in this case. Generally it is very good practice to use ID, **NOT** class.

When an HTML element is defined in JS, Jquery can then call a number of different functions on that element. One of the most important we use is the value function, which can either return the value of an html element:

let value\_of\_element = Length.val();

Or can set the value of the HTML element:

Length.val(“Hello Mars!”);

For more on JQuery visit <https://www.w3schools.com/jquery/>.

**Plotly**

Plotly is a plotting package based on the d3 graphics system. Almost every interactive plot on an Imperial Visualisation is made with plotly, due to its simplicity and great style right out of the box. After including plotly in your HTML head, you will be mostly dealing with plotly through three functions:

* **plotly.newplot**(ID,data,layout)
  + This function creates a plot in a previously unoccupied div with id equal to the ID variable. The data and layout inputs define what data is plotted and how the plot looks. Use this to initialise your plot.
* **plotly.animate**(ID, frames,animation\_attributes)
  + This function updates the data of an already existing plot in element with id = ID. The frame variable defines the new set of data. Use this for animating a constant set of points or a line, for example.
* **plotly.react**(ID, data, layout)
  + This function updates both the data and layout of an existing plot with id=ID. Use this if you need to animate axis sizes for example, or if your new data set has different dimensions than the old one.

Notice how in all three functions there is either a data or layout variable (or both). These are the two most important variables in plotly, and have a basic dictionary-like structure that is super intuitive! Let’s take a look at a data example first:

var data=[

{

type:"scatter",

mode:"lines",

x: [0,2,5,7,3],

y: [1,6,3,7,8],

line:{color:"#960078", width:3, dash: "dashed"},

},

];

As you can see data is a list of dictionaries, where the dictionaries entries are properties of a dataset; properties such as x, y, and data label. The reason data is a list is because plotly allows for the addition of multiple datasets at once onto a plot.

Now let’s look at a Layout example:

const layout = {

autosize: true,

margin: {l:30, r:30, t:30, b:30},

hovermode: "closest",

showlegend: false,

xaxis: {range: [-10,10], zeroline: true, title: "x"},

yaxis: {range: [-10,10], zeroline: true, title: "y"},

aspectratio: {x:1, y:1}

};

As you can see, layout is a single dictionary defining stylistic features of the plot, such as its size, and margins. For more on how you should style a plotly graph, visit the style guidelines page. For more on plotly, the two most important resources are:

* Function reference - <https://plot.ly/javascript/plotlyjs-function-reference/>
* Full reference - <https://plot.ly/javascript/reference/>

*Important note: Plotly can often be the reason a given page is slow or unoptimized. Look out for slow ways of updating a graph, such a plotly.purge() and plotly.newplot().*

**p5.JS**

p5.JS is a drawing tool that allows developers to create non-plot graphics (such as a mass and spring) with heavy mouse interactivity. It does this by repeatedly drawing objects on a canvas at a rate equal to 1/framerate. After p5 is imported in your html header, the first step you will want to take is creating setup and draw functions in your JS file:

function setup() {

let canvas = createCanvas(500,400);

canvas.parent(‘canvas’);

frameRate(60);

};

function draw() {

clear();

background('#ffffff');

noStroke();

fill("#A51900");

rect(mouseX, mouseY, 75,75);

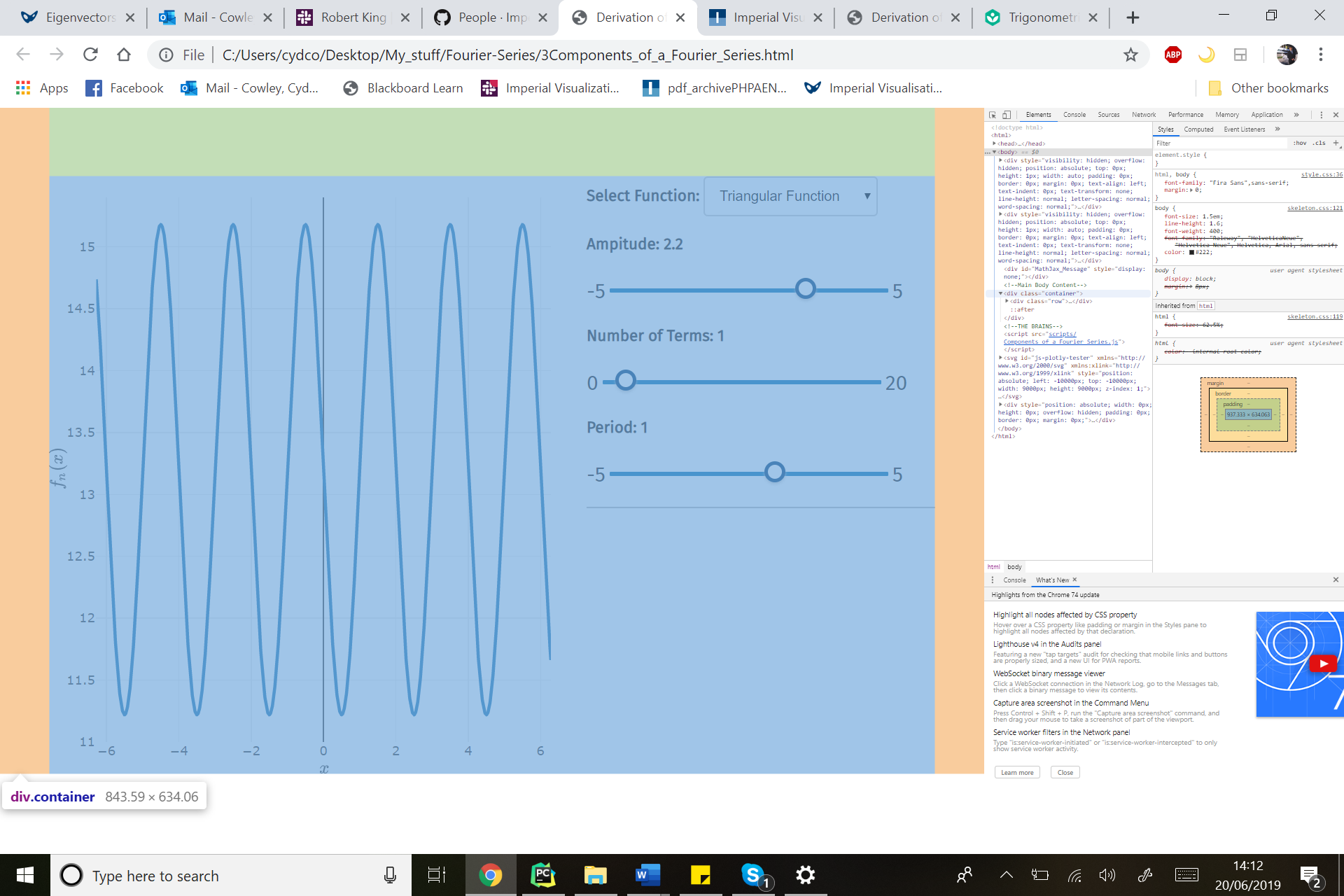
};

These two functions are automatically recognised by p5. Setup defines all the important meta-variables for p5’s drawing; variables such frame rate, and canvas size (specified in pixels). Draw is then the function that is carried out every [1/framerate] seconds. Good practice for p5 is to first clear any previous drawings, then draw your background. After that, you can draw your objects. In the example above, a brick red square is drawn at the mouse position with length and width equal to 75px.

For more on p5.JS visit <https://p5js.org/>.

**Debugging and Console**

In python debugging is easy; you press play, and if there is an error in the code your interpreter tells you exactly what and where it is. For web development, debugging is a little trickier, but the golden rule is F12 is your best friend! This hotkey (or right click + inspect for mac) brings up the browser developer tools, comprised of several different tabs. Of these, the *elements* tab is essentially your HTML/CSS debugger. If you hover over the html brought up in this tab you can see what the div you hover over looks like on the actual web page. Here we can see when the ‘container’ div is hovered over, the div and it’s appropriate padding is highlighted on the page:



The second important tab to know about is the *console* tab. This is essentially your JavaScript debugger. Any error reports in your JS code will pop up here in red, and if you click on the error it should take you to the JS code where the error was found. In addition, the analogy to the python print(“hello world”) in JS is console.log(“hello world”), where the printed output should appear in the console tab. However, when you have finished your debugging it is very important to remove all console.log commands so they do not stay in the live version of your page!

**Structuring Scripts**

A typical Imperial Visualisations JavaScript file should be split into 3 sections: the ‘Maths’, ’Interaction’, and ‘Calling’ sections:

* **Maths** – contains all the scientific code behind a visualisation. The output of this mathematical ‘black box’ should be data to be used in plotting.
  + Start by defining maths constants and variables. For example, the gravitational constant.
  + Then define maths functions and tools. For example, a function that returns the force on a mass spring system.
* **Interaction** – contains all non-mathematical functions
  + Start by defining interaction constants and variables. For example, a mass slider can be defined as M = $(#mslider)
  + Then define interactive functions. Ie, the functions that insert html values into math functions, and insert the outputted data into plotting functions.
* **Calling** – specifies which functions are called in which instance.
  + Specify which functions are called on interaction of a jquery element. For example, on “input”.
  + Specify which functions are called on load of the page.

*Important Note: It is generally recommended that the same function, ‘Update()’ is called for both inputs and onload of the page, with a variable called newpage, taking either a true or false value. This is because the only difference in between an onload and onclick function is whether to use the plotly.newplot() function or the plotly.animate() (or plotly.react()) function. Everything else is the same.*