JavaScript Applied to Maritime Design and Engineering

Henrique M. Gaspar, Norwegian Univ. of Sci. and Tech./Norway, henrique.gaspar@ntnu.no

Abstract

This paper proposes the use of JavaScript language as a key success factor when developing open and collaborative software for maritime design and engineering. Among the many script languages available for engineers, JavaScript stands out for the size of its community, easiness of learning and wide spectrum of use, from traditional web applications to server management and mobile apps. There is a resistance, however, to use and incorporate JS in engineering, usually connected to a market trust/dominance on a traditional script-like language, such as Matlab, or lack of programming skills when using a non-engineering tool to perform complex algorithms, such as using a spreadsheet-like software for calculating CFD. The topic is introduced with a discussion on the main advantages and disadvantages of JavaScript when compared to other languages in engineering, focused on speed, compatibility, user interface and usage. A basic ship design example is presented as a bridge to introduce JavaScript variables, objects, libraries and HTML document model object. Five case studies of more complex maritime software developed in JavaScript are presented: dashboards for resistance and motion calculation, a design layout tool, data-driven documents and a 3D simulator. A call for JavaScript in engineering concludes the paper.

1. The Need for Script

The idea for the topics discussed in this paper came from the need to share design and simulation developments with other academic and non-academic colleagues, as well as re-use computer-based models without worrying about proprietary software and versioning. I cannot count the amount of time wasted in meetings and classes because the results were presented in a non-interactive and closed format. Worst, when files are copied from one computer to another, libraries are missing, software versions are outdated, compilers are updated and the simulation suddenly does not run when the audience requires it. Anyone that has tried to open a spreadsheet with a macro in another computer or shared a complex Matlab code with someone else knows the feeling. Python presented itself as a solution for this problem years ago, since it is one of the best and most versatile script (high-level) languages available (www.python.org). It is free, intuitive, easy to learn and widely used in the scientific community. But for simulation and visualization in engineering it lacks important features that JavaScript (JS) has, such as simplicity to create an interactive interface, extensive 2D and 3D graphic libraries and possibility to run and share a code with no external software installation. The *internet*, or more precisely the *web,* on the other hand, has a pretty stable history of multi OS compatibility. Sites like www.efunda.com or http://www.numericjs.com/ are providing reliable online and interactive engineering/mathematic libraries for some years already.

Therefore, a disclaimer disguised as introduction: this paper will propose JS language as key success to develop open and collaborative software in engineering. It keeps an informal writing tone with unnecessary references to the first person, since that most of the examples are based on my personal experience while developing and teaching with JS as an engineer. It expects to affect two kind of readers: the ones that know about JS but never considered it a useful tool for engineers; and the engineers that do not know JS. Computer scientists are out of the scope – this paper is too simplistic for them – as well as engineers that do not develop any model-based simulation or prototype software tools – the paper has no meaning if one just press buttons in a CAD tool and drink their coffee while a 3D fined CFD/FEM mesh is being processed. The scope is even narrower, given that most of the examples from Section 4 are for maritime design and engineering. But it can be extended with no damaged information to any kind of model-based engineering problem. In other (bolder) words, I defend that, if one is considering develop an engineering software tool for data, analysis and/or simulation that must be used, interacted, shared and developed by more than one person, using JS in a web environment is a key success factor.

The background for such statement relies on the power of scripting. Script(ing) languages are often interpreted rather than compiled and typically use some sort of abstraction to hide/self-configure the internal variable types, data storage and memory management. The most well know script-like software in engineering is probably Matlab. It may not be wrong to say that every engineer graduated from and after the 1990s have some experience with the concept of scripting via Matlab, and a simple `a = 2; b = 3; a + b => 5` can be extended to non-linear dynamic simulation without (usually) worrying about compilation and memory allocation. Such simplicity, also found in other commercial software (e.g. Mathematica) or high-level languages (e.g. Python, Ruby, R, C#) is the current norm to develop model-based simulations in engineering.

The internet, however, grew in the last years much faster than these 20+ years old style of programming. And so, it changed the routine on how to present and understand the results of analyses and simulations. Interactive dashboard and visual quantification of changes are not only a reality, but expected. The tedious process of pressing the *run* button hundreds of time for *make changes here* and *see impacts there* (Nasa, 2007) can nowadays be compiled in a dashboard accessible from any pocket; pretty much every *App* in your smartphone has some sort of interactive page. Try, however, to create such engineering dashboards (Few, 2013) in a software like Matlab or Excel. Besides the cumbersome programing of outdated windows and user controls, it is practically impossible to share with others as well as control versioning without tremendous risk of loss of functionalities. JS, on the other hand, incorporates useful open source features, such as available code, traceability, reproducibility and versioning control. As engineering and software design is an interactive process, to be able to track changes and fix bugs while testing and simulating are essential in modern programming.

JS is one of the core languages of the web, together with CSS and HTML, and most modern browsers support it without plugins (Flanagan, 2011). It was released in December of 1995, made originally to control dynamically webpages, but grew in the same fast pace as the internet, being used today in pretty much every online application available, from webapps for smartphones, server management, to video game development. Besides community work, we cannot deny the role of big companies in selecting JS as main web language, such as Google's V8 JavaScript engine for compilation and execution of JS (https://developers.google.com/v8/).

In Section two I detail the disadvantages and advantages of such developments, specifically in terms of speed, compatibility, user interface and usage. Section three presents the basics of calculations in JS, from console sum to a simple web app. Section four presents a short list of examples developed by the author, while the paper concludes with a call for JS in engineering in Section 5.

2. To JavaScript or not to JavaScript

2.1. When not to JavaScript?

Before detailing the benefits of JS in engineering, let's make clear when one should NOT use JS in engineering. Such list is required to avoid frustration and angry e-mails to this author.

a) *When a spreadsheet can solve your problem in less than one afternoon*. We cannot deny the importance of spreadsheet-like programs such as Excel in engineering. It is probably the most used conceptual tool around the engineering community. Every designer/engineer has a spreadsheet *calibrated* with result from diverse simulation and experiments (e.g. sea trial). Therefore, if reliable data is available and can be gathered in less than one afternoon in just a spreadsheet, keep using it. Keep in mind that, these days there are online alternatives to this that allow for easy sharing of data, such as Google Docs (http://docs.google.com).

b) *When there is no need to use macros or VBA in Excel*. I am assuming that you use macros and VBA the same amount as I do: only when strictly necessary. Therefore, if you need a slightly more complex *for/while* loop in your spreadsheet, you will probably need to use VBA. And this will probably take more than one afternoon. And this will probably will not be compatible when Excel updates. Then Excel probably should not have been used for this task in the first place.

c) *When there is no need to share, or re-use your results*. To be fair and honest, it does not matter the time spent in coding/engineering if the results are for you only, with no need to be shared or re-used; probably you already have a code in Python or Matlab that does the job, why mess it with JS if no one will ever see or re-use this simulation?

d) *When the model does not require any graphic user interface (GUI)*. Anyone that tried to create a GUI in Pyhton, Matlab or Excel knows the limitations of these tools. One of the key positive points from using JS is the possibility to combine mathematics with dynamic elements of the HTML page. It means *any* kind of GUI, with buttons, figures, graphs as customizable as there are different pages in the net. Creativity is the limit.

e) *When one does not need to interact with the model/simulation parameters and/or data.* In line with the previous topic, interacting with data is much easier with a proper GUI. Parametric models get easy with *sliders* and real-time visualization update. Such functionalities are one of the core reasons for JS. If your model does not require the user changing variables, plotting and replotting, creating and visualizing the data generated, JS may not be necessary.

f) *When it is required to to control memory/events access and/or parallel computing*. Although modern JS engines such as Google V8 are as fast as C for certain tasks, JS is not the language of choice when one wants to easily control memory access or distribute a large task among multiple cores. True that modern JS packages take care of such events efficiently, such as Node.js (https://nodejs.org), but I would only recommend this option if you are comfortable with server management and JS. But if you do, then this paper is unnecessary.

g) *When it can be solved without coding/computer (e.g. sketching by hand/board)*. Teaching brought me many students knocking on my door with large models implemented in Matlab, Excel or even JS asking coding questions. It is very common that an answer for their problems relies not in the language of choice or ability to code, but on how the problem was modelled and the algorithm sketched. One good hour of plain paper and pencil sketches can be so efficient as hours of coding if both are done properly.

Experience shows that there is a large set of engineering cases that are not described on the excluding conditions above, and does requires mathematical models with user interactivity, efficient GUI, sharing and real time capacities. Therefore, JS benefits are described in the rest of this section.

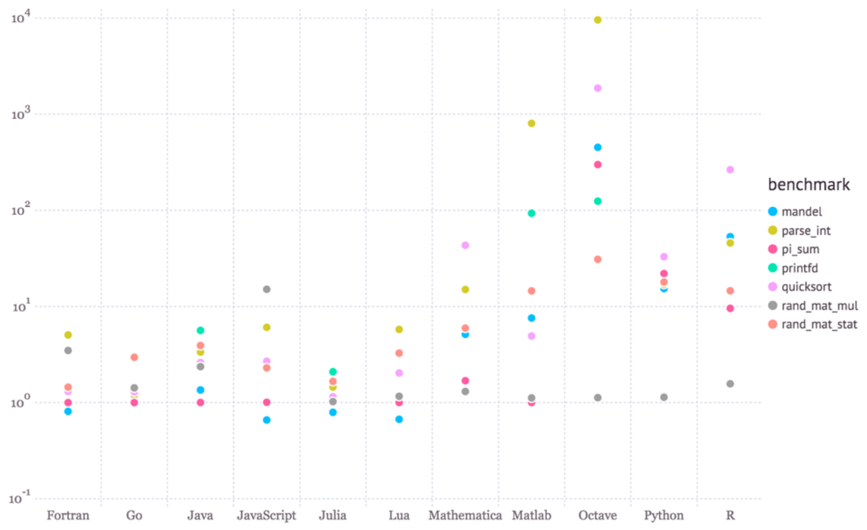
2.2 Why JavaScript

2.2.1 Speed

When I started to use JS for simulation I got impressed by the speed of calculation. Having looping *for's* and *if's* in vectors and matrices before in Matlab had brought me a certain time expectation between the *run* and the results popping out in the screen. Such expectations reached a new level when the simulation runs via a modern browser like Google Chrome in JS.

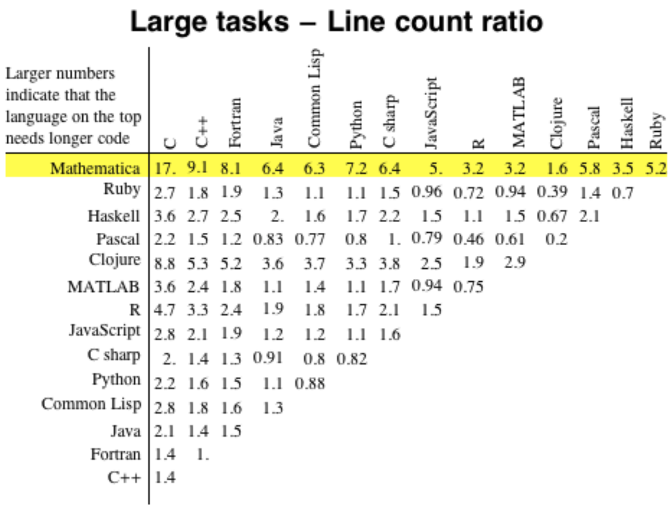
A benchmark from 2015 (https://julialang.org/benchmarks/) shows that for some basic operations, such as *parse_int,* which parses a string from a user input or a table of values given in text file, and transform this string in an integer number, JS can be over 130 times faster than Matlab. Even more impressive, complex numbers calculations such as *Mandelbrot Set* and operations like the pi sum series are faster than C, Fortran or Java (Figure 1). Given the pace of development and the fact that big companies like Goggle are behind powerful JS engines, it is not wrong to speculate that in few years the speed difference in other operations will be closer and closer to the C benchmark.

Speed to write and understand codes should also be considered. A study from 2012 (McLoone) shows that JS requires in average 3.4 times less lines of code than C and, impressively, requires 6 percent less lines in average than an equivalent large task code in Matlab (Figure 2).



	Fortran	Julia	Python	R	Matlab	Octave	Mathe-matica	JavaScript	Go	LuaJIT	Java
	gcc 5.1.1	0.4.0	3.4.3	3.2.2	R2015b	4.0.0	10.2.0	V8 3.28.71.19	go1.5	gsl-shell 2.3.1	1.8.0_45
fib	0.70	2.11	77.76	533.52	26.89	9324.35	118.53	3.36	1.86	1.71	1.21
parse_int	5.05	1.45	17.02	45.73	802.52	9581.44	15.02	6.06	1.20	5.77	3.35
quicksort	1.31	1.15	32.89	264.54	4.92	1866.01	43.23	2.70	1.29	2.03	2.60
mandel	0.81	0.79	15.32	53.16	7.58	451.81	5.13	0.66	1.11	0.67	1.35
pi_sum	1.00	1.00	21.99	9.56	1.00	299.31	1.69	1.01	1.00	1.00	1.00
rand_mat_stat	1.45	1.66	17.93	14.56	14.52	30.93	5.95	2.30	2.96	3.27	3.92
rand_mat_mul	3.48	1.02	1.14	1.57	1.12	1.12	1.30	15.07	1.42	1.16	2.36

Figure 1 – Benchmark times relative to C (smaller is better, C performance = 1.0) – https://julialang.org/benchmarks/



Large tasks – Line count ratio

Larger numbers indicate that the language on the top needs longer code	C	C++	Fortran	Java	Common Lisp	Python	C sharp	JavaScript	R	MATLAB	Clojure	Pascal	Haskell	Ruby
Mathematica	17.	9.1	8.1	6.4	6.3	7.2	6.4	5.	3.2	3.2	1.6	5.8	3.5	5.2
Ruby	2.7	1.8	1.9	1.3	1.1	1.1	1.5	0.96	0.72	0.94	0.39	1.4	0.7	
Haskell	3.6	2.7	2.5	2.	1.6	1.7	2.2	1.5	1.1	1.5	0.67	2.1		
Pascal	2.2	1.5	1.2	0.83	0.77	0.8	1.	0.79	0.46	0.61	0.2			
Clojure	8.8	5.3	5.2	3.6	3.7	3.3	3.8	2.5	1.9	2.9				
MATLAB	3.6	2.4	1.8	1.1	1.4	1.1	1.7	0.94	0.75					
R	4.7	3.3	2.4	1.9	1.8	1.7	2.1	1.5						
JavaScript	2.8	2.1	1.9	1.2	1.2	1.1	1.6							
C sharp	2.	1.4	1.3	0.91	0.8	0.82								
Python	2.2	1.6	1.5	1.1	0.88									
Common Lisp	2.8	1.8	1.6	1.3										
Java	2.1	1.4	1.5											
Fortran	1.4	1.												
C++	1.4													

Figure 2 – Average line count ratio for diverse language when programming large tasks code (McLoose, 2012)

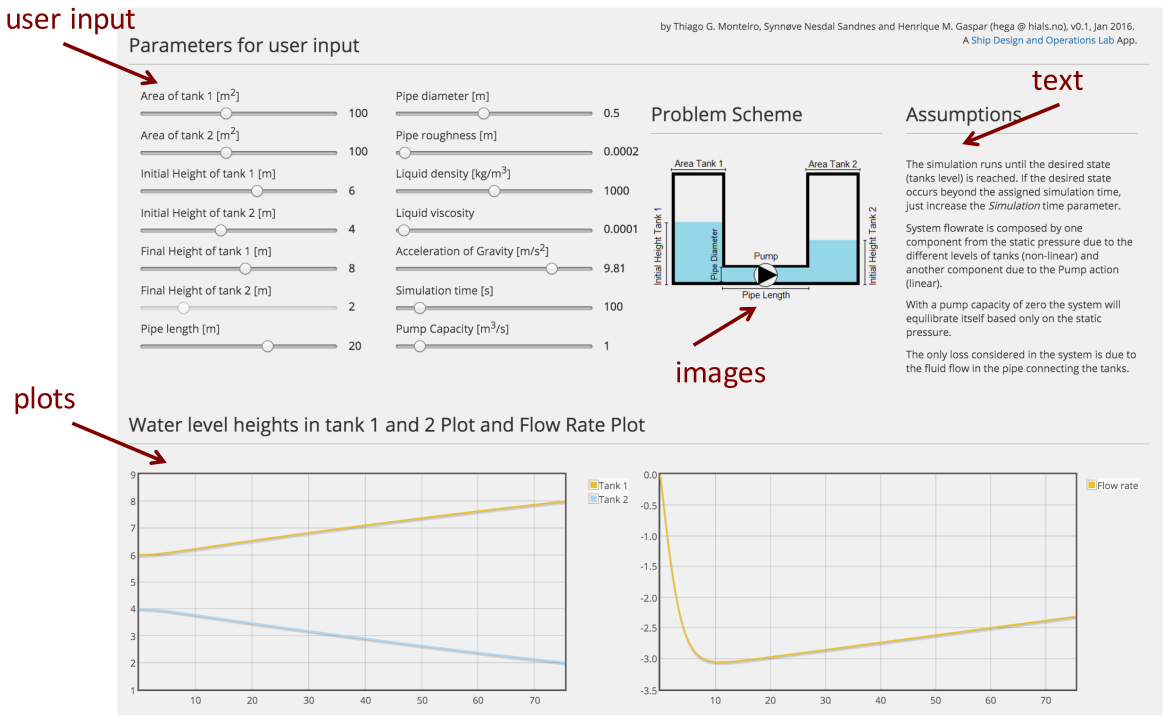
Although both benchmarks show that JS is not the fastest among all languages to process algorithms or code typing, it is so efficient as other high-level languages (either open or proprietary), with the extra functionalities from the web when combined with HTML and CSS.

2.2.2 Compatibility

Close to *universal* compatibility is the core of online applications. The idea that anyone with any modern browser can open, explore and run an engineering model in a few clicks, with no need to install or update anything is paramount. JS web applications, therefore, passes the *Shell Test.* As explained informally at UCL by my colleagues John Calleya, Michael Traut and Tristan Smith, *I think the Shell test is an imaginary person at Shell that wants to explore model data/input/output on their* *desktop without having to install anything*. In other words, web applications are a key link to show and share academic results with the society and industrial partners (exemplified by *Shell*) without the complexity that usually follows simulation models.

For the sake of exemplification, let's investigate the simulation of a coupled tanks problem. It consists of two tanks with variable areas and volumes connected between a pipe of variable length and roughness, and a pump with variable flowrate capacity. The objective is to simulate the water level heights over time, as well as the flow rate between the tanks. This would be an initial case for ballast/antiroll tank calculation, for instance. Although a simple problem, it consists of 14 inputs, with nonlinear time dependant equations due to the roughness of the pipe and viscosity of the liquid (Ingham *et al.,* 2007).

A spreadsheet model is definitively possible, with the 14 variables in 14 different cells, with a long list of time steps until the convergence of the flow[1]. Another possible option is a Matlab code, with the variable as a text input, and every *run* requires a click, plus the opening of close of multiple plots windows that always mixes with other windows. Both cases would definitively work, and such coding indeed requires less than a couple of hours. But to share the results with someone that have not been exposed to the model would require *a)* to share the Excel file and *pray* that it would work in their own version; *b)* to share the multiple *.m* Matlab files and spent other couple of hours explaining how to run and re-run the code (assuming the user has already Matlab installed); or *c)* to share a static presentation with the screenshot of few simulation plots and wait for feedback, with no interactivity.



by Thiago G. Monteiro, Synnøve Nesdal Sandnes and Henrique M. Gaspar (hega @ hials.no), v0.1, Jan 2016.
A Ship Design and Operations Lab App.

Parameters for user input

Area of tank 1 [m^2]	100	Pipe diameter [m]	0.5
Area of tank 2 [m^2]	100	Pipe roughness [m]	0.0002
Initial Height of tank 1 [m]	6	Liquid density [kg/m^3]	1000
Initial Height of tank 2 [m]	4	Liquid viscosity	0.0001
Final Height of tank 1 [m]	8	Acceleration of Gravity [m/s^2]	9.81
Final Height of tank 2 [m]	2	Simulation time [s]	100
Pipe length [m]	20	Pump Capacity [m^3/s]	1

Problem Scheme

Assumptions

The simulation runs until the desired state (tanks level) is reached. If the desired state occurs beyond the assigned simulation time, just increase the *Simulation* time parameter.

System flowrate is composed by one component from the static pressure due to the different levels of tanks (non-linear) and another component due to the Pump action (linear).

With a pump capacity of zero the system will equilibrate itself based only on the static pressure.

The only loss considered in the system is due to the fluid flow in the pipe connecting the tanks.

Water level heights in tank 1 and 2 Plot and Flow Rate Plot

Figure 3 – Coupled tanks dashboard simulation (http://www.shiplab.hials.org/app/twotanks/)

A fourth option is to create a webpage with variables as sliders, realtime plots and include a nice figure explaining the software in only one web page, a dashboard for the simulation, as observed in the screenshot from Figure 3 (http://www.shiplab.hials.org/app/twotanks/). Such interactive GUI can be opened via mobile, tablets or PCs. Sliders are very intuitive to modify variables, and the real-time update recalculates automatically every plot. It requires no compilation, no *run* button, no external

[1] How many rows is required in this spreadsheet model is an incognita, given that it can converge in few seconds when the pump has a higher flowrate or very slow if only gravity applies.

installation[2], runs direct from the browser, can be shared online (in a standard configured webserver) or private (with .HTML file and additional libraries). From the user's point of view, it requires almost no explanation when the GUI is made properly – sliders change variables, which changes the simulation and updates the plots.

2.2.3 User Interface

A clean user interface is not a merit of JS, but of the powerful combination of HTML and CSS. Different from commercial software that have a GUI constrained by cells (e.g. Excel) or predefined windows and buttons (e.g. Matlab), the browser always start from a blank page, where text, image, video, buttons, charts and most of the interactive digital commands that we are used to can be placed with large freedom on style and interface. Since 2003 CSS Zen Garden presents, for instance, the versatility of the HTML + CSS with examples of a same HTML code controlled by different CSS styles (http://www.csszengarden.com/).

Considering the example from Figure 3, we can observe that pretty much every element of the page can be customized, from colours to fonts, graphs to buttons, text, and images. A single page dashboard was the main objective in that app, with key information of the whole simulation condensed in one page. Deliberately *range slides* were used for user input, to facilitate parametric changes, instead of a traditional *text input* box. By the side of the slider is placed an image exemplifying the problem, followed by a text of the assumptions. Results of the real-time simulation are presented below the variables, with the two main plots that adjusts automatically to the maximum and minimum axis value until the convergence is reached. Most of the elements in the page, such as sliders and plots, are ready made via HTML5 or JS libraries, highly customizable for adapt to any application. Try to imagine creating the same dashboard, with the same efficiency, in any other software.

The fact that everyone knows how to use a browser is another benefit. A clean dashboard requires little or no training. Sliders instead of buttons instigates interactivity and the real-time calculation eliminate the need of pressing the *run* button every time that a parameter is changed. The concept of buttons, sliders, tabs, hyperlinks are already part of the everyday life of not only engineers, but all stakeholders from a simulation model. In this sense, to create a GUI that is understood and can be interacted by a wider range of people is an advancement.

2.2.4 Usage

Looking for similar examples in the internet is the first way to solve any difficulty while modelling. From *How to do a scatter plot in Excel* from *genetic algorithms in Python*, this is how must of us and our students looks for a solution nowadays.

Github (https://github.com/) is probably the largest repository for codes available online. A study from 2014 shows that JS is by far the most used language in Github (http://githut.info). It means software, tutorias and examples made available by millions of collaborators to re-use and contribute, with an extensive qualified community that shares its developments and results openly. Data Driven Documents library (D3 - https://d3js.org/), for instance, keeps an impressively neat page of examples, tutorials and documentation available in Github, with ready-made codes for most of the visualizations presented.

Such large community of developers means an extensive number of ready to use JS libraries available online. There is no *best choice,* since that usually the selection of a JS library is strongly connected to the needs, *Does simulation X requires a real time plot of variables or a 3D object moving in the screen, or both?* For the sake of exemplification, my personal suggestion, with list of JS libraries useful for engineers is presented as follows:

[2] But a good modern browser is recommended; and by personal experience avoid using Microsoft Internet Explorer when developing models in JS.

- Data storage: JSON (https://www.w3schools.com/js/js_json_intro.asp)[3]
- Data handling and interactivity: D3 (https://d3js.org/)
- 2D Plots: Flot (http://www.flotcharts.org/) and Chart.js (http://www.chartjs.org/)
- 3D Plots: Plotly (https://plot.ly/javascript/)
- Numerical computations: Numeric JavaScript (http://www.numericjs.com/)
- 2D Drawings (with SVG): Snap.svg (http://snapsvg.io/)
- 3D Drawings (with WebGL): Three.js (https://nodejs.org/en/)
- GPU Numerical Calculation: WebCLGL (https://github.com/stormcolor/webclgl)
- JS in the server side: Node.js (https://nodejs.org/en/)

It is worth of note to say that the *Computer Science 101* course in Stanford uses a version of JS to introduce topics of computer science to its students, such as the nature of computers and code, jargon (bits, bytes), loops, structured data and digital media. As mentioned in the prerequisites of the course *Zero computer experience is assumed beyond a basic ability to use a web browser*. Even better, the course is available online (https://cs101.class.stanford.edu/).

2.2.5 It just works

I am aware that this paper could have not been written 5-10 years ago. Most of the libraries commented in 2.2.4 were not available by then, and we did not have the powerful JS engines encoded in the browser as we have now. However, today, it just works. The fact that JS is a web language means that developers should take in consideration different devices, operating systems, browsers, versions and languages, and create standards and libraries that are functional for all of them. It is not wrong to speculate that the in the future we will not have to use a software that runs *only in Windows XP with Service Pack 2*. The reality is that we are already able to login via web in a powerful server in the cloud, allowing CFD calculations being made from a tablet and e-mailed back to you when finished (Gentzsch *et al.,* 2016).

Note that I am not affirming that every software or simulation model should be developed only in JS. But the fact that a modern browser is the new standard for user interface means that developing in JS will most likely avoid future compatibility problems between versions and operating systems. As the scope of this paper is aimed to people like me, professional engineers while amateur developers, it is a relief to realize that a code just works, does not matter if open in the *Windows 10 PC with Internet Explorer* from my boss, or in the *Macbook with Safari* from my students.

Looking at the functional side for developing model-based engineering tools JS, as a high-level language, presents many ways to solve the same problem. Data can be considered text, integers, floats, vectors (lists), matrices and objects with no need of variable declaration. Objects and prototypes are an efficient way to describe a hierarchical system, such as a ship, which proved to be very efficient when exploring a design space (Monteiro e Gaspar, 2016). Although possible to copy the similar structure of *declare variables, create function, calculate case 1 to case n, plot*, found in a spreadsheet or Matlab-like programs, one is not constrained by it, neither requires a *function with different variable names in a separated .m file* or *a VBA macro* to develop a model.

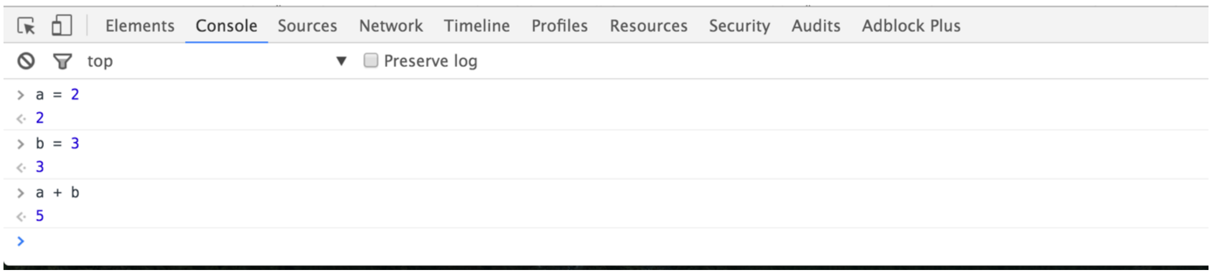
3. To JavaScript – The Basics

Many JS tutorials are available online, with Codecademy being my favourite as a self-study tool for introducing JS to students (https://www.codecademy.com/learn/javascript). The examples in these tutorials, however, are not focused in basic engineering. This section presents a simpler barge example that I have been using in the last years to introduce basic calculation in JS for my marine engineering students and colleagues. To understand the following tutorial the reader requires only knowledge in basic programming elements, such as variables, lists, functions, objects and loops. It

[3] JSON is not a library, but a syntax for storing and exchanging data, written in JS object notation. Although data can be easily converted from XML, CSV and even Excel XLXS to JSON, understanding JSON notation saves precious time when testing and using JS libraries.

starts with a basic JS console calculation (*i*) until a basic Web App with extra features (*xii*).

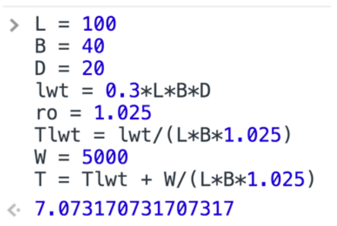
i) *Console*: Your browser is already apt to run JS code, via console, in a very similar way as the console of other high-level languages such as Python or Matlab[4]. The way to access this console changes from browser to browser, but usually it is considered a *developer tool*. When opened, it is possible to try the simple `a = 2; b = 3; a + b => 5` as observed in Figure 4.



```
> a = 2
<· 2
> b = 3
<· 3
> a + b
<· 5
>
```

Figure 4 – Google Chrome JS console with the *a*+*b* example

ii) *Barge Problem*: Imagining the simplest maritime engineering problem to start with, the draft calculation of a rectangular barge. Main variables are length (L), breadth (B), depth (D), draft (T), main deck cargo load (W), freeboard (FB) and water density (ro). Let's assume the lightweight (lwt) as a function of the cubic number, and the barge floating in sea water. Just typing the sequential variables and formulas in the console would solve automatically our problem, as observed in Figure 5.



```
> L = 100
  B = 40
  D = 20
  lwt = 0.3*L*B*D
  ro = 1.025
  Tlwt = lwt/(L*B*1.025)
  W = 5000
  T = Tlwt + W/(L*B*1.025)
<· 7.073170731707317
```

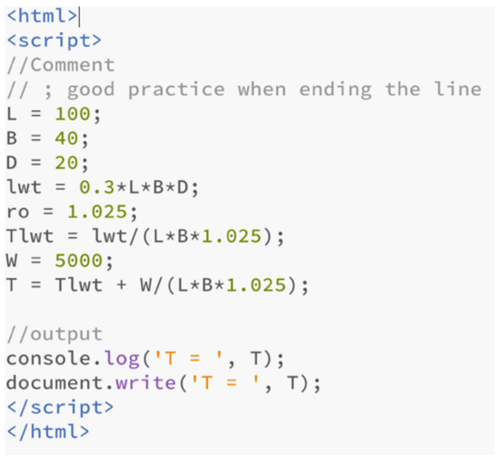
Figure 5 – Screenshot of the barge problem solved in the browser JS console

iii) *Editor*: Console is a useful tool to debug your code, but a good editor is fundamental. Many free choices are available. Brackets (http://brackets.io/) is my actual choice, with many useful features as autocompletion, Github connection and a pleasant colouring scheme.

iv) *Editing*: pure JS code is not computed in the browser, it needs to be written in the HTML file (browser readable) among the tags *<script></script>* or included in it as external library. Therefore, bringing the barge problem to a standalone file, made in the editor, would add few lines, as well as better understanding of our problem. The same code from Figure 5 is presented in Figure 6, typed in the editor and saved as *.html* file. Running this code means opening the .html file that it was saved as, giving that this code would work in any modern browser, with no need of installing any additional software. The *console.log()* function is used to print the result (variables and objects) in the console.

v) *HTML*: Besides printing outputs in the console, we can get any variable printed in the main page, via the *document.write()* function. The *Document* interface is an object model (DOM), and represents any web page loaded in the browser, serving as an entry point into the web page's content (https://developer.mozilla.org/en-US/docs/Web/API/Document). Most of the element and attributes contained in a *Document* can be tagged, identified and modified (Figure 7), from text to figures and interactive graphs. It means that we can put tags and ids in the elements of the page, and modify these elements via JS, updating a result after a calculation or when a parameter is changed, for instance[5].

[4] In strict terms Matlab is not really considered a high-level language.
[5] To know more about the potential of handling the HTML document object model (DOM), elements, attributes and events, start with the good and reliable W3S tutorials (https://www.w3schools.com/js/js_htmldom.asp)



```
<html>
<script>
//Comment
// ; good practice when ending the line
L = 100;
B = 40;
D = 20;
lwt = 0.3*L*B*D;
ro = 1.025;
Tlwt = lwt/(L*B*1.025);
W = 5000;
T = Tlwt + W/(L*B*1.025);

//output
console.log('T = ', T);
document.write('T = ', T);
</script>
</html>
```

Figure 6 – Screenshot of the barge problem in the editor, using the colour scheme of an .html file[6].

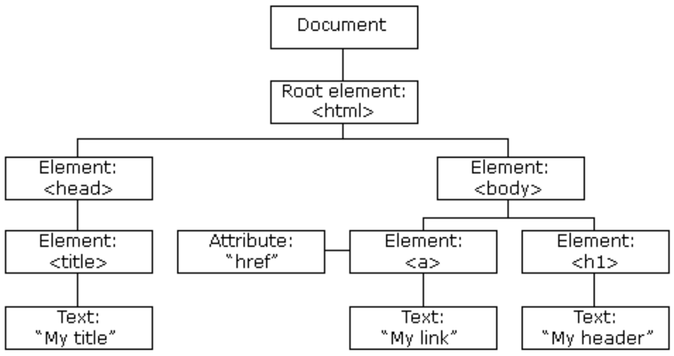
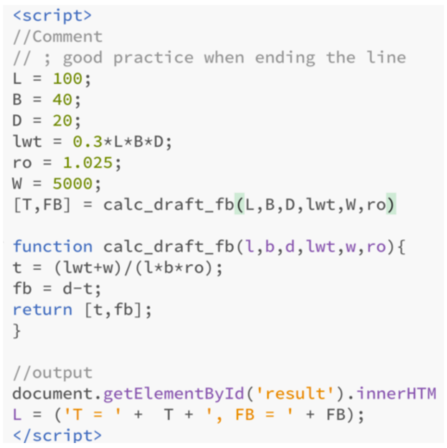


Figure 7 - The HTML Document Object Model (DOM) three of objects (https://www.w3schools.com/js/js_htmldom.asp)

vi) *Functions*: Creating a function in JS follows similar syntax as other high-level languages: it requires a declaration, input of variables or objects, calculation and output of results. Figure 8 presents the code from Figure 8 with a *calc_draft()* function, which receives *L, B, D, lwt, W* and *ro* and outputs *T* and *FB*.



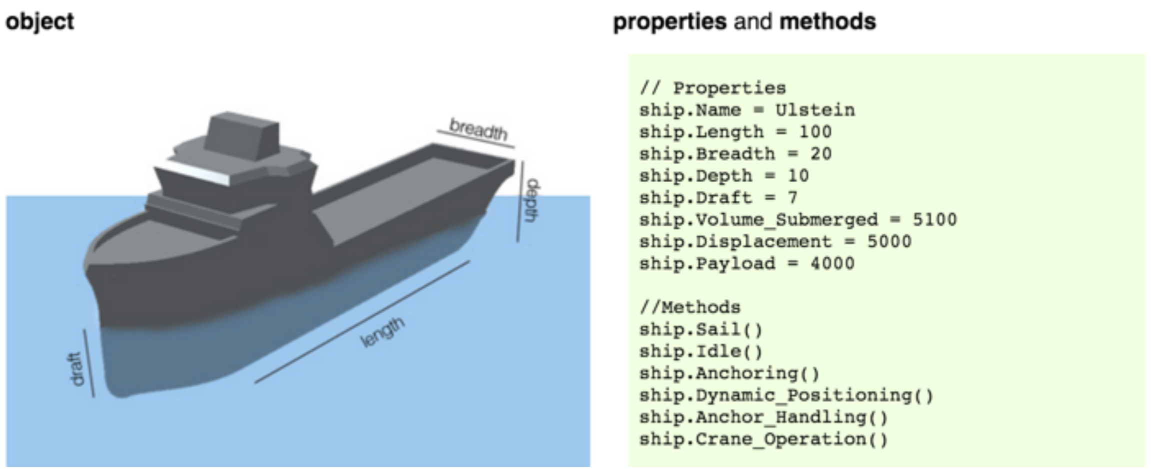
```
<script>
//Comment
// ; good practice when ending the line
L = 100;
B = 40;
D = 20;
lwt = 0.3*L*B*D;
ro = 1.025;
W = 5000;
[T,FB] = calc_draft_fb(L,B,D,lwt,W,ro)

function calc_draft_fb(l,b,d,lwt,w,ro){
t = (lwt+w)/(l*b*ro);
fb = d-t;
return [t,fb];
}

//output
document.getElementById('result').innerHTM
L = ('T = ' +  T + ', FB = ' + FB);
</script>
```

Figure 8 – *Calc_draft* function to calculate draft *T* and freeboard *FB* for the barge problem

vii) *Objects*: The barge can be defined as an object, with properties and methods. Such taxonomy is one of the key elements of JS to handle complex hierarchical structures as ships and its simulation. A well-defined object can be used in different calculations, for simulation and visualization of complex

[6] Good practice recommends changing *<html></html>* for *<!doctype html>*.

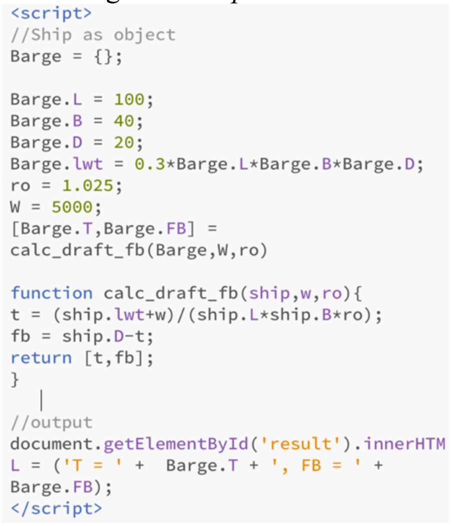
engineering models. Figure 9 presents the idea of a ship as an object, with properties like *Name* and *Length,* and methods such as *Sail()* and *Anchoring()*.



```
// Properties
ship.Name = Ulstein
ship.Length = 100
ship.Breadth = 20
ship.Depth = 10
ship.Draft = 7
ship.Volume_Submerged = 5100
ship.Displacement = 5000
ship.Payload = 4000

//Methods
ship.Sail()
ship.Idle()
ship.Anchoring()
ship.Dynamic_Positioning()
ship.Anchor_Handling()
ship.Crane_Operation()
```

Figure 9 – a ship as a JS object, with properties and methods.

Similarly, the barge from our problem can be defined as an object in the code described in Figure 10. It consists of the user input properties *L, B, D*, while *T* and *FB* are calculated via the function *calc_draft()*. Note that the function receives now an object *ship* as input, rather than the variables separated, and uses the properties of this generic *ship* for its calculation.

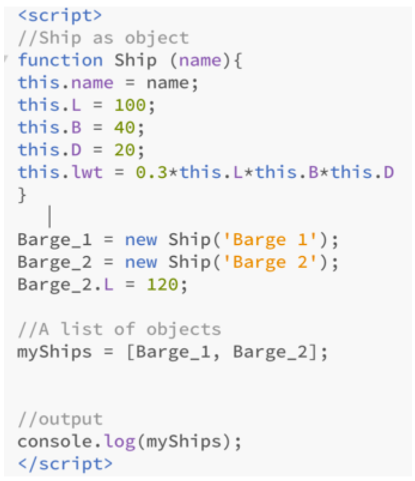


```
<script>
//Ship as object
Barge = {};

Barge.L = 100;
Barge.B = 40;
Barge.D = 20;
Barge.lwt = 0.3*Barge.L*Barge.B*Barge.D;
ro = 1.025;
W = 5000;
[Barge.T,Barge.FB] =
calc_draft_fb(Barge,W,ro)

function calc_draft_fb(ship,w,ro){
t = (ship.lwt+w)/(ship.L*ship.B*ro);
fb = ship.D-t;
return [t,fb];
}

//output
document.getElementById('result').innerHTM
L = ('T = ' +  Barge.T + ', FB = ' +
Barge.FB);
</script>
```

Figure 10 – The barge presented as an object *Barge*.

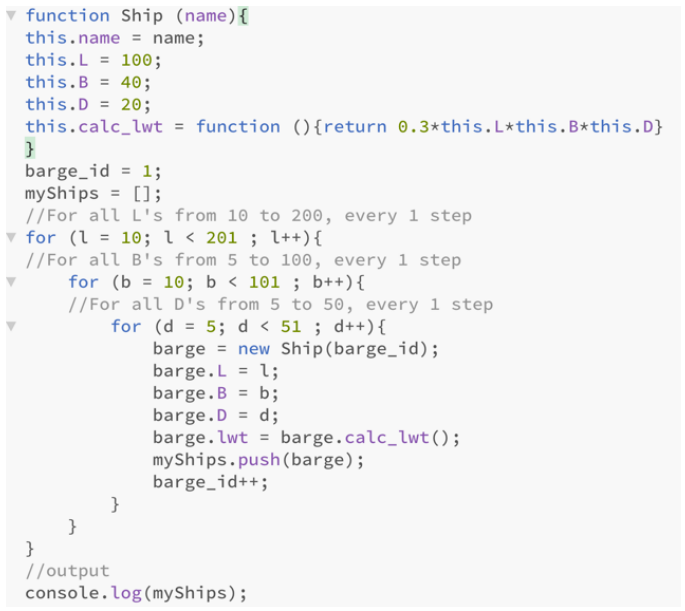
viii) *Prototype*: every JS object has a prototype, which is also an object. One way is to understand prototype is as a function that constructs a new object, which inherits the properties of the prototype constructor. Let's imagine *Ship()* as a prototype, able to create any object with properties *L, B, D* and *lwt*, such as *Ship X* and *Ship Y* instances. Both instances would inherent the properties and methods of *Ship()*, but can also be modified and include new properties and methods if necessary. Figure 11 presents a simple code for a *Ship()* prototype with two instances, *Barge_1* and *Barge_2*, with the last having its property length *L* modified from 100 to 120. A list called *myShips[]* is created, containing both instances.

ix) *Design Space*: With a constructor in place, we can start to create a preliminary design space, using a series of *for* loops in order to vary *L, B* and *D*. Figure 12 exemplifies this process for creating a design space with 769526 unique instances, varying in a single unit step the properties length (*L,* from 10-200), breadth (*B,* from 10-100) and depth (*D,* from 5-50). The whole design space is saved in the list *myShips[]*.



```
<script>
//Ship as object
function Ship (name){
this.name = name;
this.L = 100;
this.B = 40;
this.D = 20;
this.lwt = 0.3*this.L*this.B*this.D
}

Barge_1 = new Ship('Barge 1');
Barge_2 = new Ship('Barge 2');
Barge_2.L = 120;

//A list of objects
myShips = [Barge_1, Barge_2];


//output
console.log(myShips);
</script>
```

Figure 11 – *Ship()* as prototype to construct instances



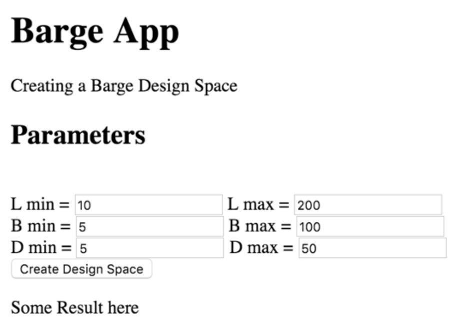
```
function Ship (name){
this.name = name;
this.L = 100;
this.B = 40;
this.D = 20;
this.calc_lwt = function (){return 0.3*this.L*this.B*this.D}
}
barge_id = 1;
myShips = [];
//For all L's from 10 to 200, every 1 step
for (l = 10; l < 201 ; l++){
//For all B's from 5 to 100, every 1 step
    for (b = 10; b < 101 ; b++){
    //For all D's from 5 to 50, every 1 step
        for (d = 5; d < 51 ; d++){
            barge = new Ship(barge_id);
            barge.L = l;
            barge.B = b;
            barge.D = d;
            barge.lwt = barge.calc_lwt();
            myShips.push(barge);
            barge_id++;
        }
    }
}
//output
console.log(myShips);
```

Figure 12 – A series of *for* loops to create a design space with 799526 unique instances based on the *Ship()* constructor

x) *User Interface*: So far, we have been calculating our barge in the console, as we would do in Matlab. To bring the potential of the web we need to use the browser as GUI, creating a *web app.* It requires the understanding that two different languages are interacting between each other to control the main document (DOM): HTML and JS.

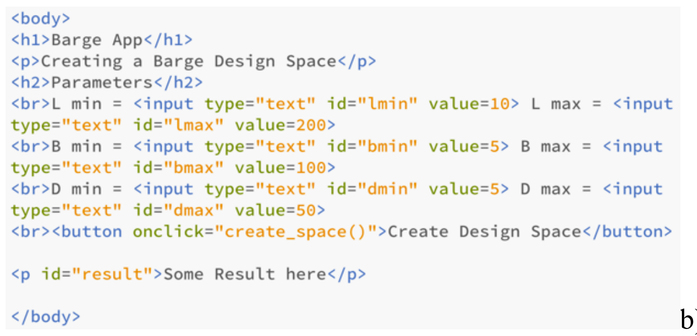
The HTML part of the code between the tags *<body></body>* controls what we see and interact in the main body of the DOM. A text between the tags *<h1></h1>* will be interpreted as a title, while *<input type=´´text´´>* shows in the screen a box where the user can input any text or number. A button for calling a calculation function is possible and presented, but practically unnecessary, given that each of elements can trigger actions on certain events, such as *onchange, onclick* or *onmouseover.* Keep in mind that each element requires a unique *id*, since the JS part of the code will interact with it, reading input and writing outputs. A very simple GUI with a title and the parameters presented in *ix* is observed in Figure 13a while Figure 13b shows the equivalent HTML code.

xi) *Web App*: To create an interactive web app the JS code must interact with the HTML elements. For this simple *Barge App* case it means reading the user input, calculate all the possible designs and print info about them back in the document under the div identified as *result.* The code presented in Figure 14b parses the string of each of the input boxes from the HTML, creates the design space, and print the whole list of unique designs on the document, via the *getElementbyid()* method. The final web app with the design space calculated after user input is presented in Figure 14a.



```
Barge App

Creating a Barge Design Space

Parameters

L min = 10    L max = 200
B min = 5     B max = 100
D min = 5     D max = 50
Create Design Space

Some Result here
```

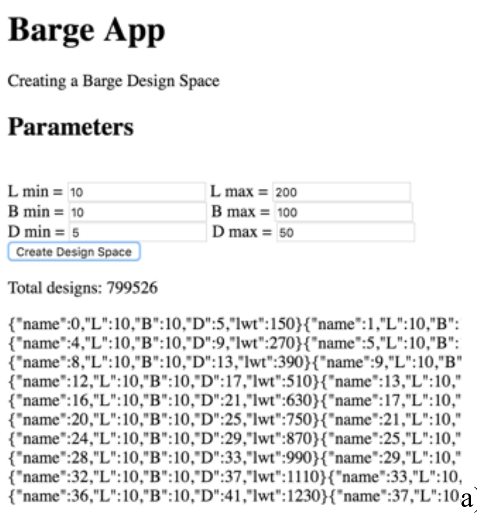
a)



```html
<body>
<h1>Barge App</h1>
<p>Creating a Barge Design Space</p>
<h2>Parameters</h2>
<br>L min = <input type="text" id="lmin" value=10> L max = <input
type="text" id="lmax" value=200>
<br>B min = <input type="text" id="bmin" value=5> B max = <input
type="text" id="bmax" value=100>
<br>D min = <input type="text" id="dmin" value=5> D max = <input
type="text" id="dmax" value=50>
<br><button onclick="create_space()">Create Design Space</button>

<p id="result">Some Result here</p>

</body>
```

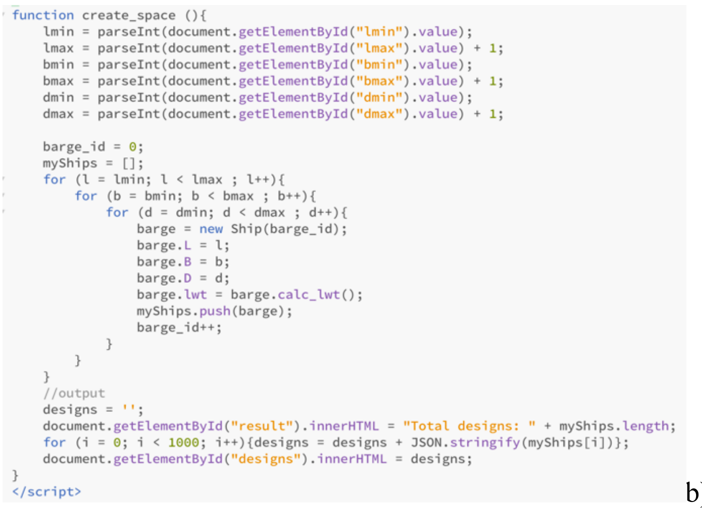
b)

Figure 13 – Basic GUI for a *Barge App* (a) and its equivalent HTML code (b)



```
Barge App

Creating a Barge Design Space

Parameters

L min = 10    L max = 200
B min = 10    B max = 100
D min = 5     D max = 50
Create Design Space

Total designs: 799526

{"name":0,"L":10,"B":10,"D":5,"lwt":150}{"name":1,"L":10,"B":
{"name":4,"L":10,"B":10,"D":9,"lwt":270}{"name":5,"L":10,"B":
{"name":8,"L":10,"B":10,"D":13,"lwt":390}{"name":9,"L":10,"B"
{"name":12,"L":10,"B":10,"D":17,"lwt":510}{"name":13,"L":10,"
{"name":16,"L":10,"B":10,"D":21,"lwt":630}{"name":17,"L":10,"
{"name":20,"L":10,"B":10,"D":25,"lwt":750}{"name":21,"L":10,"
{"name":24,"L":10,"B":10,"D":29,"lwt":870}{"name":25,"L":10,"
{"name":28,"L":10,"B":10,"D":33,"lwt":990}{"name":29,"L":10,"
{"name":32,"L":10,"B":10,"D":37,"lwt":1110}{"name":33,"L":10,
{"name":36,"L":10,"B":10,"D":41,"lwt":1230}{"name":37,"L":10
```

a)



```javascript
function create_space (){
    lmin = parseInt(document.getElementById("lmin").value);
    lmax = parseInt(document.getElementById("lmax").value) + 1;
    bmin = parseInt(document.getElementById("bmin").value);
    bmax = parseInt(document.getElementById("bmax").value) + 1;
    dmin = parseInt(document.getElementById("dmin").value);
    dmax = parseInt(document.getElementById("dmax").value) + 1;

    barge_id = 0;
    myShips = [];
    for (l = lmin; l < lmax ; l++){
        for (b = bmin; b < bmax ; b++){
            for (d = dmin; d < dmax ; d++){
                barge = new Ship(barge_id);
                barge.L = l;
                barge.B = b;
                barge.D = d;
                barge.lwt = barge.calc_lwt();
                myShips.push(barge);
                barge_id++;
            }
        }
    }
    //output
    designs = '';
    document.getElementById("result").innerHTML = "Total designs: " + myShips.length;
    for (i = 0; i < 1000; i++){designs = designs + JSON.stringify(myShips[i])};
    document.getElementById("designs").innerHTML = designs;
}
</script>
```

b)

Figure 14: Barge App GUI with 799526 designs (a) and JS code interacting with the HTML elements via *getElementbyid()* method (b)

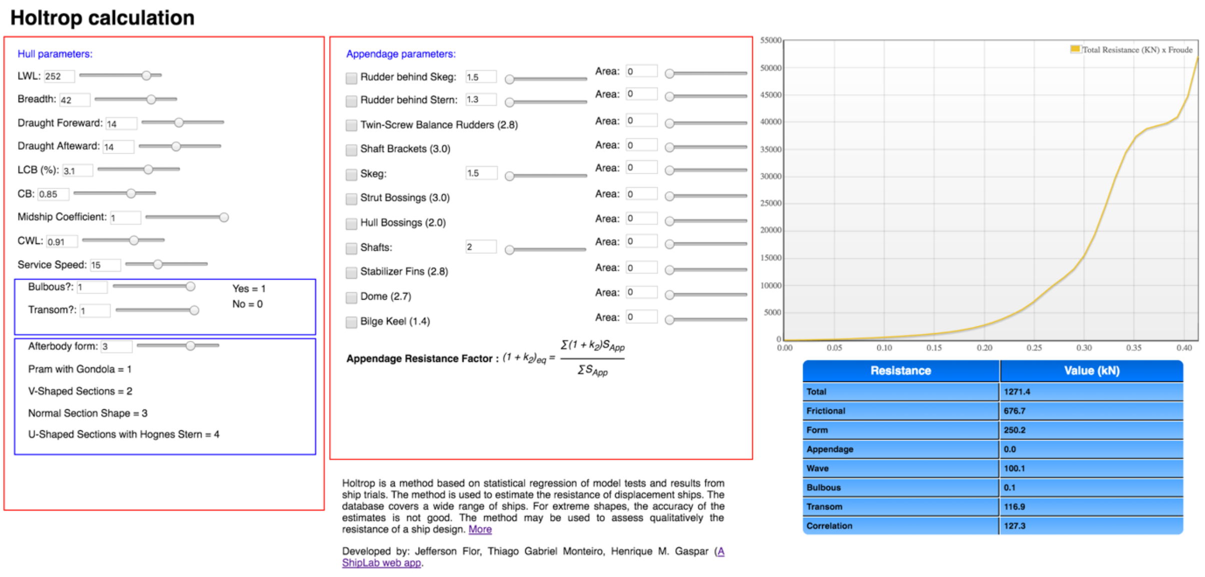
xii) *Extra Features*: Adding extra features is a natural step further. JS offers so many options that, rather than technology availability, user requirements and creativity are the main constraints. Improvements in the GUI are done by properly use of CSS and ready-made template libraries, such as the *Bootstrap* framework (http://getbootstrap.com/). Graphs and plots can be inserted, as well as any other digital features observed in web pages, such as range sliders, buttons, video, sound, 2D and 3D drawings. This interactivity is the main bonus of JS, since no other language provides so much freedom and examples when manipulating the DOM document to create useful GUIs.

4. Maritime Design and Engineering

This section presents short a list of examples of web apps and simulations developed in the last years by the author at the Ship Design and Operations Lab at NTNU in Ålesund (www.shiplab.hials.org). Most of the examples are available online, with code and algorithms free to update and reuse.

i) *Ship Resistance*: Figure 15 presents a dashboard to calculate in real-time the total resistance of a ship via the Holtrop & Mennen method (1982; Holtrop 1984). It was mainly developed by a master student in a couple of weeks (J. Flor, 2016, http://shiplab.hials.org/app/holtrop/).

ii) *Ship Motion*: Estimation of ship motions via closed-form expressions is a powerful method to derive frequency responses for the wave-induced motions for monohull ships (Jensen *et. al.,* 2004). Figure 16 presents a dashboard to calculate motion and acceleration using closed-form expressions, developed also by a master student (S. Andrade, 2015, www.shiplab.hials.org/app/shipmotion/).


Holtrop calculation

Hull parameters:

LWL:	252
Breadth:	42
Draught Foreward:	14
Draught Afteward:	14
LCB (%):	3.1
CB:	0.85
Midship Coefficient:	1
CWL:	0.91
Service Speed:	15
Bulbous?:	1
Transom?:	1

Yes = 1
No = 0

Afterbody form: 3

Pram with Gondola = 1
V-Shaped Sections = 2
Normal Section Shape = 3
U-Shaped Sections with Hognes Stern = 4

Appendage parameters:

Rudder behind Skeg:	1.5	Area: 0
Rudder behind Stern:	1.3	Area: 0
Twin-Screw Balance Rudders (2.8)		Area: 0
Shaft Brackets (3.0)		Area: 0
Skeg:	1.5	Area: 0
Strut Bossings (3.0)		Area: 0
Hull Bossings (2.0)		Area: 0
Shafts:	2	Area: 0
Stabilizer Fins (2.8)		Area: 0
Dome (2.7)		Area: 0
Bilge Keel (1.4)		Area: 0

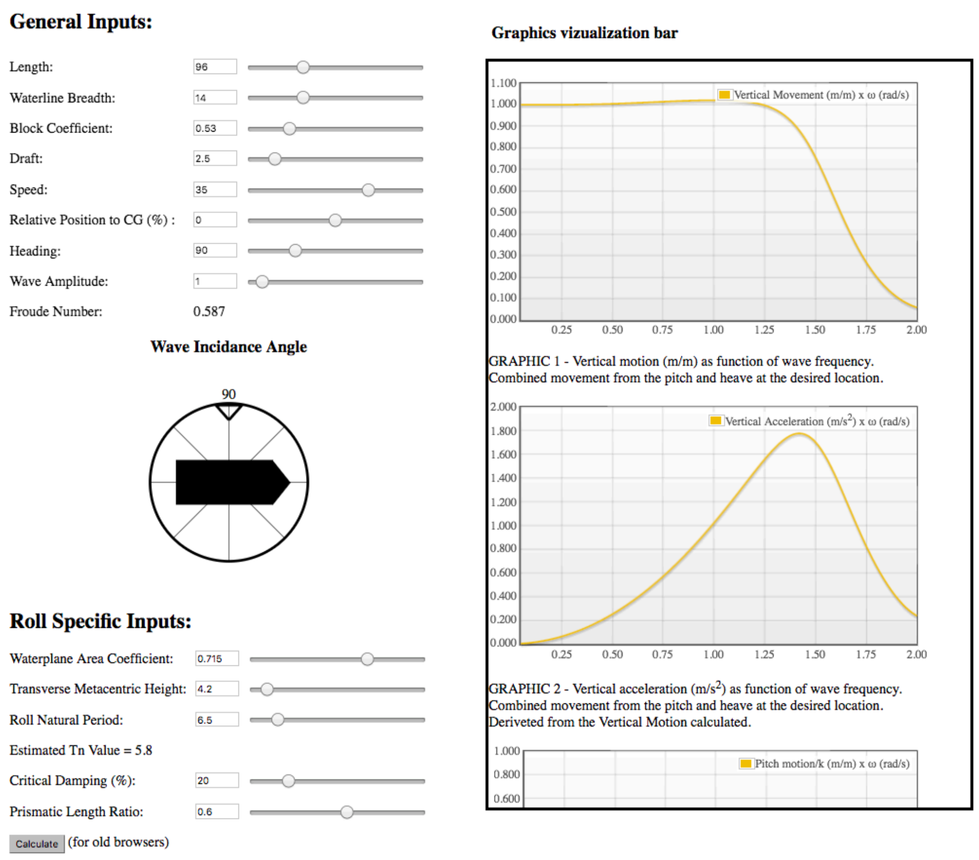
Appendage Resistance Factor : $(1+k_2)_{eq} = \dfrac{\Sigma(1+k_2)S_{App}}{\Sigma S_{App}}$

Holtrop is a method based on statistical regression of model tests and results from ship trials. The method is used to estimate the resistance of displacement ships. The database covers a wide range of ships. For extreme shapes, the accuracy of the estimates is not good. The method may be used to assess qualitatively the resistance of a ship design. More

Developed by: Jefferson Flor, Thiago Gabriel Monteiro, Henrique M. Gaspar (A ShipLab web app.

Resistance	Value (kN)
Total	1271.4
Frictional	676.7
Form	250.2
Appendage	0.0
Wave	100.1
Bulbous	0.1
Transom	116.9
Correlation	127.3

Figure 15 – Dashboard for total resistance calculation via Holtrop & Mennen method.


General Inputs:

Length:	96
Waterline Breadth:	14
Block Coefficient:	0.53
Draft:	2.5
Speed:	35
Relative Position to CG (%) :	0
Heading:	90
Wave Amplitude:	1
Froude Number:	0.587

Wave Incidance Angle

Roll Specific Inputs:

Waterplane Area Coefficient:	0.715
Transverse Metacentric Height:	4.2
Roll Natural Period:	6.5
Estimated Tn Value = 5.8	
Critical Damping (%):	20
Prismatic Length Ratio:	0.6

Calculate (for old browsers)

Graphics vizualization bar

GRAPHIC 1 - Vertical motion (m/m) as function of wave frequency.
Combined movement from the pitch and heave at the desired location.

GRAPHIC 2 - Vertical acceleration (m/s^2) as function of wave frequency.
Combined movement from the pitch and heave at the desired location.
Deriveted from the Vertical Motion calculated.

Figure 16 – Ship motion calculation via closed-form expressions.

iii) *Layout Tool*: A proof of concept tool to demonstrate that it is possible to quickly develop a web-based app for handling ship design layout during conceptual phase is presented in Figure 17. The app reads from a database (.csv file) the general arrangement information as *blocks*, plot them into a grid (Figure 17a) and evaluate their positioning, attributes, neighbours and connections. A connection wheel plots the relationship between blocks (Figure 17b) to quantify physical interfaces. The total work for develop this tool was approximatively 14 hours, including concept development, coding, debugging, examples, layout detailing and text writing.

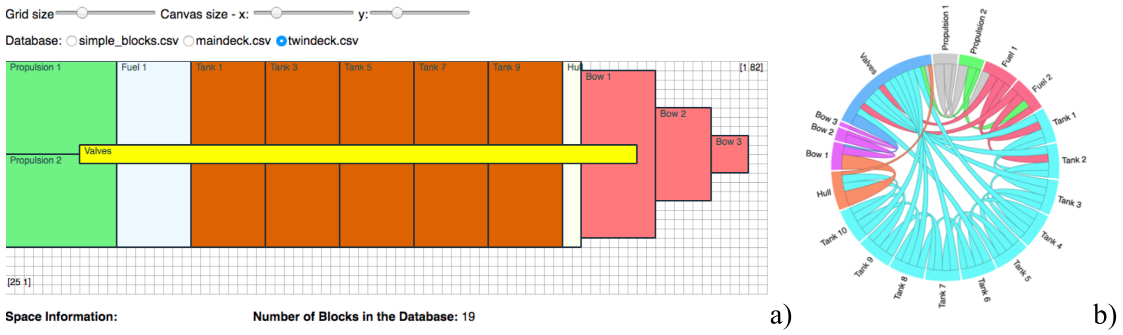


Figure 17 – Ship design layout tool (http://uscience.org/files/grid/ H. Gaspar, 2015)

iv) *Data-driven documents (D3):* Figure 18 presents five D3 examples applied to the conceptual ship design process, based on Gaspar *et al.* (2014) and Calleya *et. al*. (2016). All examples are done in JS, and visualized in a standard .html file via browser.

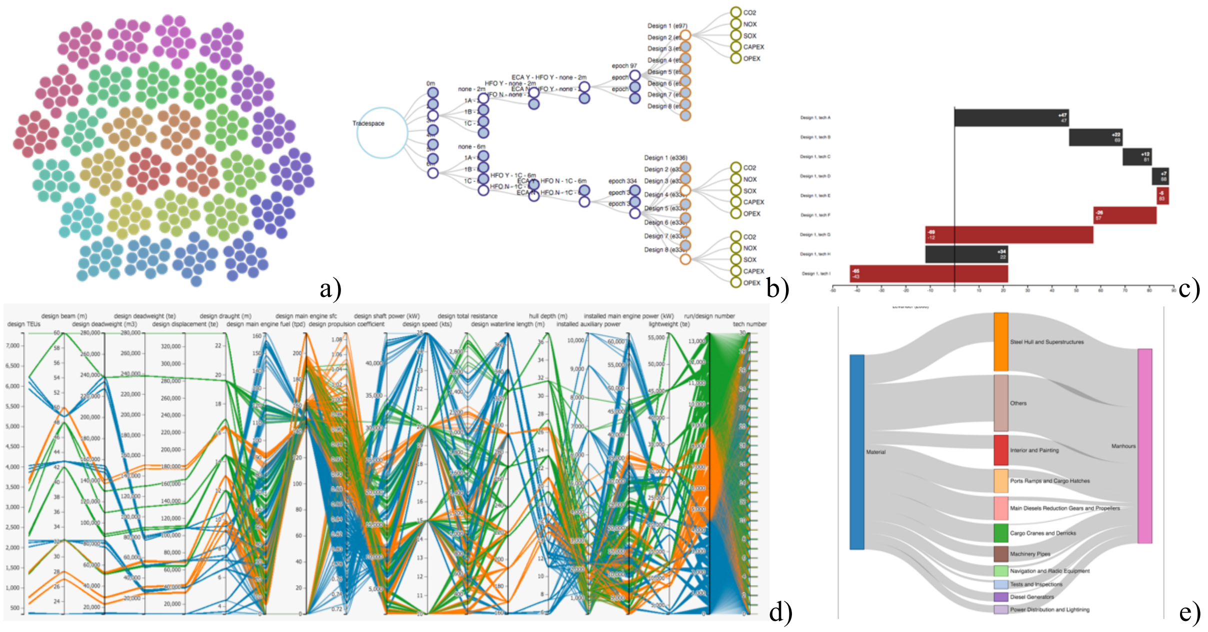


Figure 18 – D3 examples applied to ship design: a) clustering of 360 designs; b) tree layout to comprise 336 scenarios and 13440 simulation results; c) A waterfall chart to quantify effect of design options; d) Parallel coordinates chart applied to 13392 design; e) Sankey diagram representing the relationship between material and hours cost (based on Gaspar *et al.* (2014) and Calleya *et. al*. (2016))

v) *3D Simulation*: Figure 19 presents screenshots from the DNVGL COMPIT 2016 award winner simulator, developed by Chaves and Gaspar (2016). The tool is based on modular design theory, closed form-expressions and WebGL libraries (http://uscience.org/compit2016/).

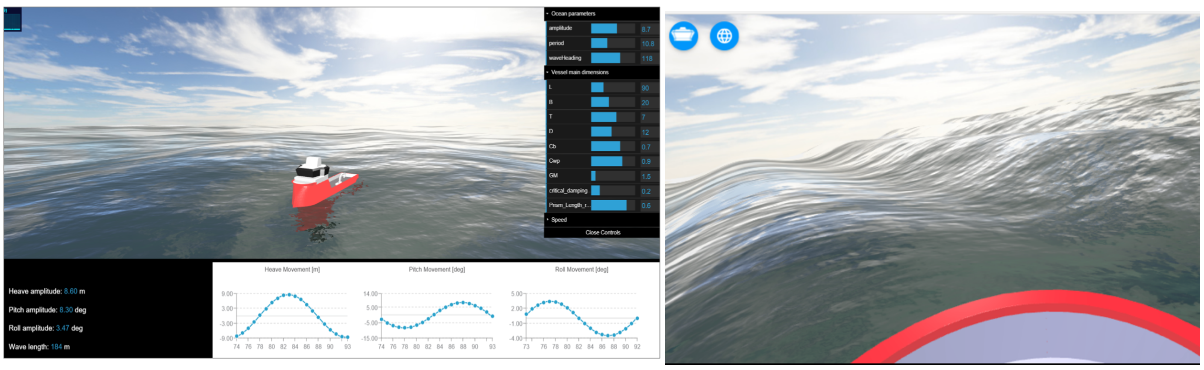


Figure 19 – Screenshots from the 3D *Ship Motion Simulator* (Chaves & Gaspar, 2016)

5. Call for JavaScript

I close this paper with a call for my colleagues and students to consider developing future engineering analysis and simulation in JS. I believe that, combined with web elements, no other framework will allow so much continuing collaboration and re-use of codes and libraries as JS. Even for more advanced applications JS is becoming a reliable option, with online compilers such as *WebAssembly* coming as standard feature soon in modern browsers, which will allow C and C++ compilation direct from the client (http://webassembly.org/).

Regarding JS and the ship design community, other examples rather than the ones presented by this author are yet very seldom. To instigate more users, a repository with ship related JS codes, developed by me and my students, is being organized (www.vesseljs.org). Monteiro is a pioneer of this process with a *Vessel.js* library already in place (Monteiro 2016; Monteiro and Gaspar, 2016). Resistance and motions methods as the ones discussed in Section 4 are already available as JS functions, as well the ship as object via system based design.

Research is also being developed on testing the limits of the tool. The simulator from Figure 19, for instance, is being tested with multibody analysis, and nowadays 10^3 different floating bodies is an acceptable order of magnitude when using parametric equations or pre-calculated CFD analyses. GPU calculation for particles via *WebCLGL* is also another instigating topic, with potential to future real-time fluid dynamics simulation.

Acknowledgements

The author holds currently an associated professor position at the Department of Ocean Operations and Civil Engineering at NTNU (Ålesund, Norway), and has no commercial or professional connection with the software and companies cited. Statements on usability and performance reflects solely my opinion, based on personal experience, with no intention to harm or diminish the importance of current commercial state-of-the-art engineering tools.

References

CALLEYA, J; PAWLING, R; RYAN, C; GASPAR, H.M.; (2016) *Using Data Driven Documents (D3) to explore a Whole Ship Model*, System of Systems Engineering Conference (SoSE), 2016 11th

CHAVES, O.; GASPAR, H.M (2016), *A Web Based Real-Time 3D Simulator for Ship Design Virtual Prototype and Motion Prediction*, 15th COMPIT Conf., Lecce

FEW, S. (2009), *Now you see it*, Analytics Press

FLANAGAN, D. (2011), *JavaScript: The Definitive Guide,* O'Reilly & Associates

GASPAR, H.M; BRETT, P.O.; EBRAHIM, A.; KEANE, A. (2014), *Data-Driven Documents (D3) Applied to Conceptual Ship Design Knowledge,* 13th COMPIT Conf., Redworth

GENTZSCH, W., PURWANTO, A.; REYER, M. (2016), *Cloud Computing for CFD based on Novel Software Containers*, 15th COMPIT Conf., Lecce

HOLTROP, J. (1984), *A Statistical Reanalysis of Resistance and Propulsion Data*, Int. Shipbuilding Progress 31

HOLTROP, J.; MENNEN, G. (1982), *An approximate power prediction method,* Int. Shipbuilding Progress 29

INGHAM, J.; DUNN, I.J.; HEINZLE, E.; PRENOSIL, J.E.; SNAPE, J.B. (2007), *Chemical Engineering Dynamics,* Wiley

JENSEN, J.J.; MANSOUR, A.E.; OLSEN, A.S. (2004), *Estimation of ship motions using closed-form expressions*, Ocean Engineering 31/1, pp.61-85

McLOOSE, J. (2012), *Code length measured in 14 languages*, Wolfram http://blog.wolfram.com/2012/11/14/code-length-measured-in-14-languages/

MONTEIRO, T. (2016), *A Knowledge-Based Approach for an Open Object Oriented Library in Ship Design*, MSc Thesis, NTNU, Trondheim

MONTEIRO, T.; GASPAR, H. M. (2016), *An Open Source Approach for a Conceptual Ship Design Tools Library*, 10th HIPER Conf., Cortona

NASA (2007) *NASA Systems Engineering Handbook,* NASA