Vessel.js: an open and collaborative ship design object-oriented library

H. M. Gaspar
Department of Ocean Operations and Civil Engineering, NTNU (Ålesund, Norway)

1 VESSEL.JS: AN OPEN SOLUTION FOR DATA-DRIVEN SHIP DESIGN

Effective use of ship value chain data is a key factor for successful projects during the preliminary stages of design. Ship Data can be represented in two complimentary categories, top-down and bottom-up. The first comprises data from formulas, regressions, previous designs and parametric studies based on existing solutions, such as the one commented in Parsons (2011), most of Watson and Gilfillan (1976) and recently Roh and Lee (2018). Such top-down approach to collect data usually finds a practical and consensual solution at short time and cost but lacks in innovation and carries on the bias and out of date insights from previous designs. Bottom-up data is connected to specific key elements and subsystems that directly affects one or more phases of the value chain. Bottom-up data presents different taxonomies for vessel division and understanding, such as in methods like systems-based ship design (Levander, 2011) and design building block (Andrews, 2003). Bottom up data is the starting point for innovative and technological break-through designs, but suffers from the lack of knowledge connected to the uncertainty of one-of-a-kind projects.

Data-driven ship design methods (Gaspar, 2018) aims to connected both data categories, presenting a unique data structure able to consolidate top-down and bottom-up approaches during early stages of design, combining value chain data, from concept to operation. Commercial PDM/PLM software promises such integration, but it usually comes with a high integration cost and limited freedom to customize libraries and calculations.

Vessel.js is a JavaScript library for data-driven design combining conceptual ship design with a web-based object-oriented approach. *Vessel.js* represents the vessel as an JSON object, which is used to simulate different functionalities and behaviours. Currently, the library includes methods for hydrostatic and stability calculations, as well as parametric hull resistance and closed-form seakeeping equations, but it intends to grow to incorporate linear and non-linear hydrodynamics and structural models. *Vessel.js* has the ambition to be an open and collaborative web-based library toolset for ship design data, freely inspired by data-driven documents library (Bostock *et. al.*, 2011), able to handle relevant information about the vessel systems and key components, as well as relevant parametric data when this is within the useful range, with a special flexibility for managing different taxonomies and attributes. The data-driven framework seen here is yet in its early stages and relies heavily on community to grow. The current paper presents the initial stages of *Vessel.js* development, introducing its design principles, basic functionalities, examples and a call for collaborators. The library is free, open and developed by the Ship Design and Operation Lab at Norwegian Univ of Science and Technology (NTNU) (http://vesseljs.org).

2 JAVASCRIPT FOR DESIGN AND ENGINEERING

2.1 *Why JavaScript*

The internet grew in the last years much faster than the engineering programming style (Gaspar, 2017). 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; pretty much every *App* in a 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 cost-free share with others as well as control versioning without tremendous risk of loss of functionalities. JavaScript (JS), on the other hand, incorporates useful open source features, such as available code, traceability, reproducibility and versioning control. As engineering and design is an interactive process, to be able to track changes and fix bugs while testing and simulating are essential in modern programming. Moreover, 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.

Gaspar (2017) claims five benefits for developing engineering applications in JS: speed, usage, compatibility, user interface and reliability, summarized as follow.

2.2 *Speed*

JS engines from web browsers, such as Google V8, are extremely optimized to performance in most of modern machines, and most of the cases provides a great cost-benefit in terms of simplicity versus computational power. 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 in JS than C, Fortran or Java. 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. McLoose (2012) 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. 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.

2.3 *Usage*

A study from 2017 shows that JS is by far the most used language in Github (https://octoverse.github.com/), with over 2.3M active repositories, way ahead of the second place (Python, 1M repositories) and Java (986k repositories). Such large community means that many software, tutorials and examples made by thousands collaborators are available 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.

2.4 *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 (Gaspar, 2017). In other words, web applications are a key link to show and share academic results with the society and industrial partners without the complexity that usually follows simulation models.

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.

2.5 *User Interface*

JS interactive graphical user interfaces (GUIs) can be visualized 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 installation, 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.

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. Pretty much every element of the page can be customized, from colours to fonts, graphs to buttons, text, and images.

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.

2.6 *It just works*

I am aware that this JS would not have been the language of choice to develop a ship design library 5-10 years ago. Most of the functionalities and libraries discussed 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). As the scope of this paper is aimed to people like the author and his students, 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 the boss, or in the Macbook with Safari from the students (Gaspar, 2017).

3 VESSEL.JS LIBRARY

3.1 *Overview – Ship as an Object*

One of the fundamental concepts from object-oriented programming is the inheritance concept. Usually in a class-based programming language, objects are instances of classes, from which they can inherit properties and functions. In JavaScript objects inherit from special objects called prototypes. JS also allows encapsulation, enclosing all the functionalities of an object within that object so that the object's methods and properties are protected from the rest of the application, allowing particular properties and functions to be specific within the boundary of the object. These two concepts allow the build of applications with reusable code, scalable architecture, and abstracted functionalities expected in Vessel.js idea (Monteiro and Gaspar, 2016).

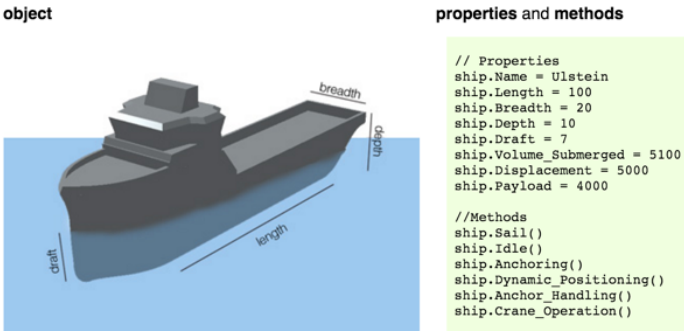


Figure 1 – A ship as a JS object, with properties (e.g. name, length) and methods (e.g sail, idle) as basis for the Vessel.JS library structure (Gaspar, 2017).

A vessel and its subsystems can be thus defined as objects (within objects), with general (pre-defined by the library) and customized (defined case to case) properties and methods. Such freedom in establishing a taxonomy is one of the key elements of *Vessel.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 engineering models. Figure 1 presents the idea of a ship as an object, with properties like *Name* and *Length*, and methods such as *Sail()* and *Anchoring()*. This basic principle is incorporated in a generic virtual prototype model, detailed as follows.

3.2 *Virtual Prototype Model applied to Ship Design*

Inspiration for the fundamental elements required for an efficient web-based object-oriented virtual prototype library were taken from He *et al. (*2014). The authors states that a successful virtual prototype model is composed of other three subsequently models: entities (EM), states (SM) and processes (PM). *Entities* are related to the physical and logical components of the product, including 2D and 3D models, mostly reflecting the relation between basic structure and geometric information of each part; *States* are connected on how the entity behaves under changes in internal and external stimuli, which we interpreted as the main analyses; Lastly, *processes* incorporate entities and states in longer periods of time, that is, the accumulation of different state models for different conditions.

A preliminary application of this framework in ship design was sketched by Fonseca and Gaspar (2015) and recently updated (Fonseca *et. al.,* 2018). The model considers the ship and its systems/parts as entities, while common analyses are tackled by a math library which updates the the state model. Processes over time are considered missions, which can be as simple as a manoeuvring simulation or as complex as the fuel consumption simulation of the whole operational lifecycle. The interaction between these three models is observed in Figure 2.

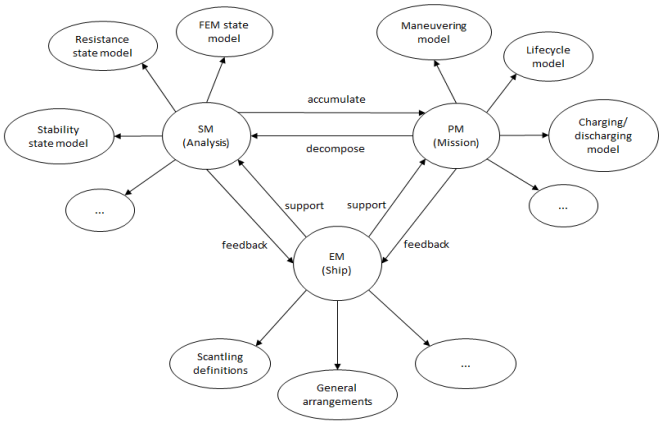


Figure 2 – Virtual prototype model applied to ship design and simulation (Fonseca and Gaspar, 2015; adapted from He et al.*, 2014).*

The main structure of the *Vessel.js* library presented in the rest of this Section tackles mainly the EM and SM elements. Modelling processes (PMs) usually requires extensive external information, not always connected to the structure of the ship and its systems, and it must be organized case by case. A proposal for a stepwise PM, from a design space to a multibody simulation is exemplified in Section 5.

3.3 *Vessel.JS structure*

The basic structure of *Vessel.js* (Figure 3) incorporates so far three types of fundamental elements: *classes, fileIO* and *math*. These elements are able to model most of the ship related entities (EM) and analyse internal behaviour of a ship under different states (SM), such as fuel consumption and hydrostatics calculations.

classes is the heart of the library, with seven sub-libraries able to handle most of the ship design data including entities and internal states. *fileIO* contains functions for file exchange, such as load and download a pre-existing ship file. *math* aggregates a library of formulas and regressions that can be used for analysis and design, from simple assumptions like the sum of an area, to more complex non-linear formulas like hull resistance via *Holtrop* (Holtrop, 1984) and closed-form functions for seakeeping (Jensen *et al.*, 2004).

Moreover, the structure was designed with the open and collaborative mind-set, therefore it is flexible enough to allow the incorporation of other elements, such as new formulas, coefficients and regressions in *math*, new file formats in *fileIO* and ship elements in *classes*, as well as modification of the existing one, such as different assumptions for calculating GM in *Hull.js*.

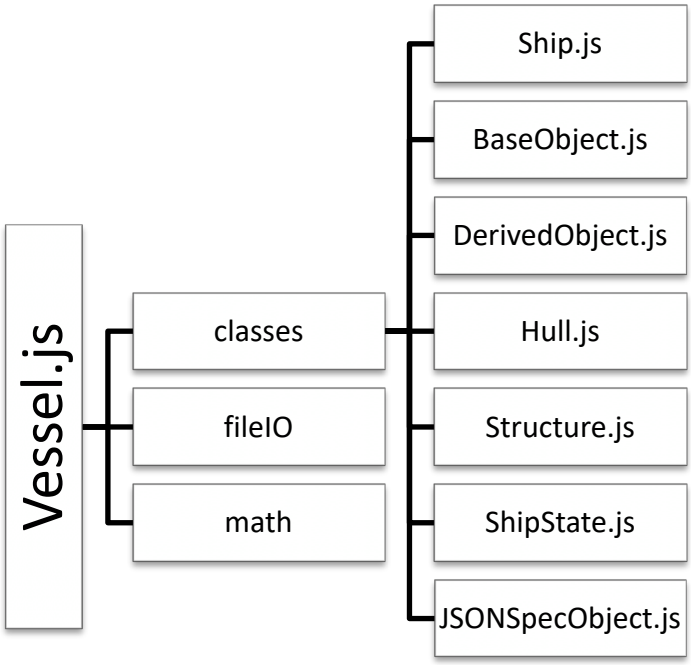


Figure 3- Vessel.js structure, with core classes, fileIO and math.

3.4 *classes*

classes serves to properly handle different entities and states data from the ship. On the current version of the library (v0.13) there are seven key classes, as follow.

BaseObject.js: the image of every class object defined in the library, except the hull and structure. In other words, it constructs the library of physical entities (systems/subsystems) that construct the ship. It can be defined from a simple component (e.g propeller) to the whole machinery room. The main idea is that a base object can be served as basis for many derived objects. For instance, a base object *tank* defines a fixed height, volume, lightweight and 3D model for a given type of tank, as well as capabilities specific to a tank, such as pressure limit and manufacturer. Later, this tank can be copied several times in the ship, with different fluids and levels inside the ship. *BaseObject.js* can also contain a library of several engines, while only one will be instanced to the ship, adding standard attributes for this engine (area, volume, 3D model) as well as specific capabilities for engines (such as specific fuel consumption, fuel type and efficiency curves). A database of systems and components, with 3D models and internal methods, is expected to be available in future versions of the library.

DerivedObject.js: classes for individual ship component entities, derived from *BaseObject.js*. Every design is composed by a set of derived objects. In other words, every physical object considered in a *Ship{}* object must have a derived object. *DerivedObject.js* is the class that defines properties for

individual component such as mass, center of gravity and tank filling. A vessel with six tanks based on one instance from *tank* from the base object, for instance, could have six derived objects (e.g. *tank01, tank02, tank03, tank04, tank05, tank06)*, with each object having different specific properties, such as fluid density and deck position.

Hull.js: represents the geometric ship hull, currently based on a traditional table of offsets. Rather than dealing the hull as a derived object, this class handles hull as an object with very specific properties, containing internal methods to calculate most of its attributes, such as volume, displacement and common coefficients (e.g. C_B, C_P, C_M). This encapsulates most of hull data and methods inside the class, allowing easy access of its properties in other parts of the code. This class also includes methods for GM calculation and for truncating hull visualization according to a given draft. A database of hulls is expected to be available in future versions of the library.

JSONSpecObject.js: hidden base class for objects that are constructed from a literal object, (or optionally from a JSON string). Currently used to load and export objects in JSON format. The default implementation is now at preliminary stage.

Structure.js: defines structural specific objects, such as decks and bulkheads. It is in incipient stage of development but intends to include all relevant structural elements of a ship, including methods to handle longitudinal, transversal and local structural analyses. Future developments of this class intends to be connected to the *math* part to incorporate classification society rule-check and basic structural analysis. A database of structural elements and basic configurations is also aimed in the long term.

Ship.js: this is the ship itself, defined by a set of elements from other classes, that is, one hull (from *Hull.js*), one division of decks and bulkheads (from *Structure.js*) and all general arrangements (from *DerivedObject.js*). The Ship class will collect the analysis inherently to the vessel, performed by the library as they are implemented, such as GM considering hull (hydrostatic) and objects (weights). A database of ships with different hull and arrangements is expected to be available in future versions of the library.

ShipState.js: stores ship states for simulation calculations. This can be simple change in states, such as fuel level, heading and CG, to more complex state changes, such as powering under DP and demanding marine operations, checking simultaneous state of several derived objects, such as tanks, cranes and engine. *ShipState.js* currently accounts for load state, or the states of objects in the ship.

The object state assignments also have assignment rules based on groups and IDs, facilitating operations filtered by different taxonomies. A group of *baseObjects* could for instance be a category including all tanks that carry a given compound, regardless of their size and shape. The same goes for *derivedByGroup*. With this, there would be five types of assignments, where assignments of subsequent types override assignments of previous types:

- *common*: All objects.
- *baseByGroup*: Applies to every object that has its base object's property "group" set to the given name.
- *baseByID*: Applies to all objects that have base object consistent with the given ID.
- *derivedByGroup*: Applies to every object that has its property "group" set to the given name.
- *derivedByID*: Applies only to the object with given ID.

The caching and version control of ship states is incomplete at this stage but intends to control states at different frequencies for complex maritime simulation, for instance*, engine under DP* must be updated in 1000Hz, while seakeeping converges in 100Hz and fuel consumption can be checked for every 1Hz. In this sense, complex time-domain simulations can be handled efficiently for every object of the ship.

3.5 *fileIO*

Class with pre-defined functions for handling file exchange, currently using JSON format. It is composed of three main methods:

Vessel.browseShip(): handy function for letting the user load a ship design from a local file)

Vessel.downloadShip(ship): very simple function to download of the specification of a given ship design in JSON format.

Vessel.loadShip(url, callback): function for loading a ship design from file

The *fileIO* class can be extended to allow exchange of files in any known open format, such as XML or comma-separated values (CSV).

3.6 *math*

math handles common ship-design functions used for calculations inside the core classes, but that are

not necessary connected to one specific object. It consists of generic formulas such as moments of inertia calculation, as well as empirical formulas which requires inputs from different objects, such as parametric weight calculations. When one uses the work of Parsons (2011) or recently Roh and Lee (2018), most of the formulas presented there could be incorporated in this class. Currently, the class has the minimum required formulas to calculate basic hydrostatic and few parametric assumptions. Some of these formulas is exemplified as follows:

areaCalculations.js: calculation of sectional areas and moments of inertia.

combineWeights.js: returns mass and CG for a combination of weight arrays.

interpolation.js: linear and bilinear interpolation helpers. Also includes *bisectionSearch()*, a function that takes a sorted array as input, and finds the last index that holds a numerical value less than, or equal to, a given value. Returns an object with the index and an interpolation parameter *mu* that gives the position of value between *index* and *index+1*.

parametricWeightParsons.js: this function estimates the structural weight of the hull for simple draft calculations. This includes the weight of the basic hull to its depth amidships. It is based on Watson and Gilfillan (1976). This function exemplifies parametric equations.

vectorOperations.js: some small helpers for operations on 3D vectors. A vector is defined as an object with properties x, y and z.

volumeCalculations.js: volume calculation for vessel displacement based on numerical integration of sectional areas. Uses as input the table of offsets.

3.7 *Supporting Libraries*

Diverse supporting JS libraries are used to create the examples presented in the next section. This exemplifies the power of developing in JS. 3D plots, for instance, are handled by *WebGL* via *Three.js* library, while change in parameters are easily handled via *dat.gui*.

A list of credited supporting libraries (used in examples only) follows:

Bootstrap: http://getbootstrap.com/ - used for creating responsive sites, that works well in PCs, tablets and mobile phones.

jQuery: https://jquery.com/ - used to simplify client-side scripting in HTML.

three.js: https://threejs.org/ - used for 3D plotting and 3D user interface,

dat.gui: https://github.com/dataarts/dat.gui/ - a lightweight controller library for changing design and object parameters

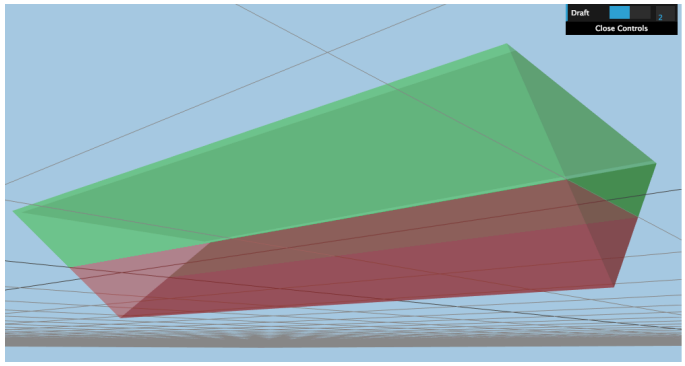
renderjson: http://caldwell.github.io/renderjson - used to output design files in JSON.

D3: http://d3js.org - data-driven documents used to 2D plots and visualization of vessel data (previously applied in ship design by Gaspar *et al.,* 2014).

4 VESSEL.JS EXAMPLES

4.1 *A very simple hull*

A very simple prismatic hull was created to verify the basic hydrostatic of the library. This exemplify the most basic top-down approach: a hull is given, and hydrostatic properties are calculated for each given draft and angle of list (Figure 4).



Hydrostatic Properties

Variable	Results
Draft Amidships (m)	2.00
Volume of Displacement (m^3)	100.00
Waterline Length (m)	20.00
Maximum Waterline Breadth (m)	5.00
Waterplane Area (m^2)	100.00

Cb	0.50
Cp	1.00
Cwp	1.00
KB (m)	1.33
LCB (m)	10.00
LCF (m)	10.00
BMt (m)	2.08
BMl (m)	33.33

Figure 4 – Prismatic hull and its hydrostatic properties calculated by Vessel.js

The only info required as input in this case is the table of offsets of the hull. In the case of the prismatic hull observed in Figure 4 , the JSON object required for the calculation would contain simply:

```
"hull": {
            "attributes":  {"LOA":  20,  "BOA":  10,
"Depth": 4, "APP": 0,},
            "halfBreadths":   {"waterlines":   [0,1],
"stations": [0,1], "table": [ [0,0],[1,1]]},
            "buttockHeights": {}
        }
```

The benefit of a HTML GUI can already be seen in this very simple example: the draft of the ship is changed by a slide in the top right corner of the

screen, very intuitively. Automatically all hydrostatic properties are updated when the value is changed.

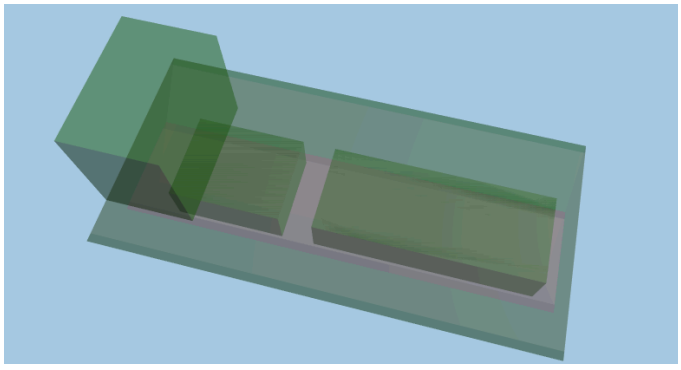
4.2 *A very simple ship – from top to bottom*

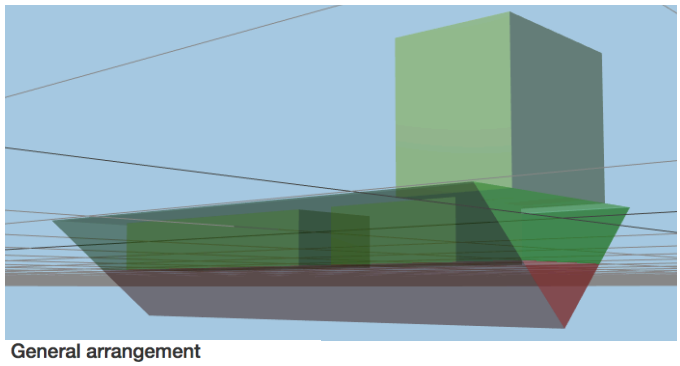
The next step in our simple case is to start to transform the hull into a ship, including physical entities to it. Each of these objects must have basic standard properties, such as positon, area, volume, weight and 3D shape. For the sake of exemplification, lets create a base object tank, and derive two ship objects from it, placing each of them in different coordinates and different decks.

```
  baseObjects": [
  {
   "id": "TankL ",
   "boxDimensions": {
                   "length": 4,
                   "breadth": 6,
                   "height": 5
               },
   "file3D": "tank1.stl",
   "baseState": { "fullness": 0},
   "weightInformation": {"contentDensity": 0, "volu-
meCapacity": 120, "lightweight": 15000}
  }

  "derivedObjects": [
   {"id": "Tank1",
    "baseObject": "TankL",
    "affiliations": {"Deck": "WheelHouseTop", "SFI":
"101"},
    "referenceState": {"xCentre": 2, "yCentre": 0,
"zBase": 4}
   },
   {"id": "Tank2",
    "baseObject": "TankL",
    "affiliations": {"Deck": "BridgeDeck", "SFI":
"102"},
   "referenceState": {"xCentre": 5, "yCentre": 0,
"zBase": 2}
  },
```

This is just an example, and more info is required as input for a proper object-oriented analysis, such as deck and bulkheads information (included in *Stucture.js)* as well as a fullness state for every tank in *ShipState.js.* A weight for the hull is also expected if the same hydrostatic properties observed in Figure 4 are going to be calculated. A simple GUI for adding block, as well as the effects in the 3D plot of a ship containing a prismatic hull and three objects is observed in Figure 5.





General arrangement

Specification

ID	Lenght	Breadth	Height	Deadweight
Block0	4 m	6 m	5 m	15000 kg
Block0	5 m	4 m	2 m	14000 kg
Block2	10 m	4 m	2 m	56000 kg

Calculations

Variable	Hand value	Calculated value
Deadweight (kg)	85000.000	85000.000
Total area (m2)	84	84
Total Volume (m3)	240	240

Weight Definition

Create block | Delete block

ID	z Base (m)	Weight (kg)
Block0	4.00	15000.00
Block1	2.00	14000.00
Block2	2.00	56000.00

Figure 5 – Adding blocks to the basic hull, exemplifying a possible transition to top-down to bottom-up.

Note that, with hull and entities as elements, draft is no longer an input, but an output of the code, with objects (hull, blocks) as inputs. This may be the simplest example differencing between top-down and bottom up approaches. Adding objects starts to change the approach from a known behaviour *Draft is 2m,* to a question *What is the draft for this general arrangement?* given that, with weight, area and volumes, the draft is consequence of the equilibrium equations rather than initial input. This trigger other functions in the library to calculate the same hydrostatic properties and other simple analysis, given that VCG and LCG is known, such as GM calculation, as observed in Figure 6. The same objects could be encapsulated by a different hull, creating a new design, new GM.

Calculations

Variable	Results
Hull Weight (kg)	197836.84
Blocks Weight (kg)	85000.00
Displacement (kg)	282836.84
Draft (m)	3.32
BMt (m)	3.46
BMl (m)	20.06
GMl (m)	19.00
GMt (m)	2.40
KB (m)	2.22
KG (m)	3.27

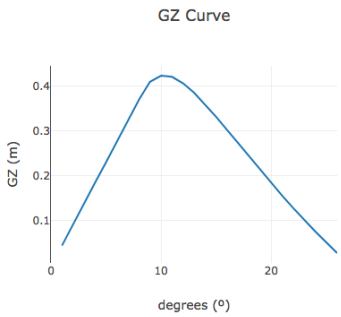


Figure 6 – KG is provided by the objects in the ship, allowing a simplified GM calculation.

4.3 *PSV Example*

A more detailed platform supply vessel (PSV) is currently being used as example for more advanced calculations. The vessel consists of 106 derived objects. Different systems are incorporated in the design, such as propulsion, cargo, deck and accommodation. The vessel can also be organized in different

taxonomies, with each of the entities being able to be included in one or more classification. In this way, the object *Tank1,* for instance, can be part of the *Cargo System*, as well as the *Midship Volume*, and the library would allow a colour representation of each of these different taxonomies.

Each of the objects is also able to receive one or more external files as attributes, for instance a 2D drawing, bill of materials or 3D file. In the current example, each derived object is connected to a library of 3D base objects, and they are rendered according to the information available in the library: if the object has an external 3D .STL file, this will be rendered in real time, otherwise a basic shape can be choose, such as cuboids and cylinders, to be plotted in the height and width defined. The example in Figure 7 shows the PSV, with the rendering of the 106 objects via their 3D .STL files or basic shapes.

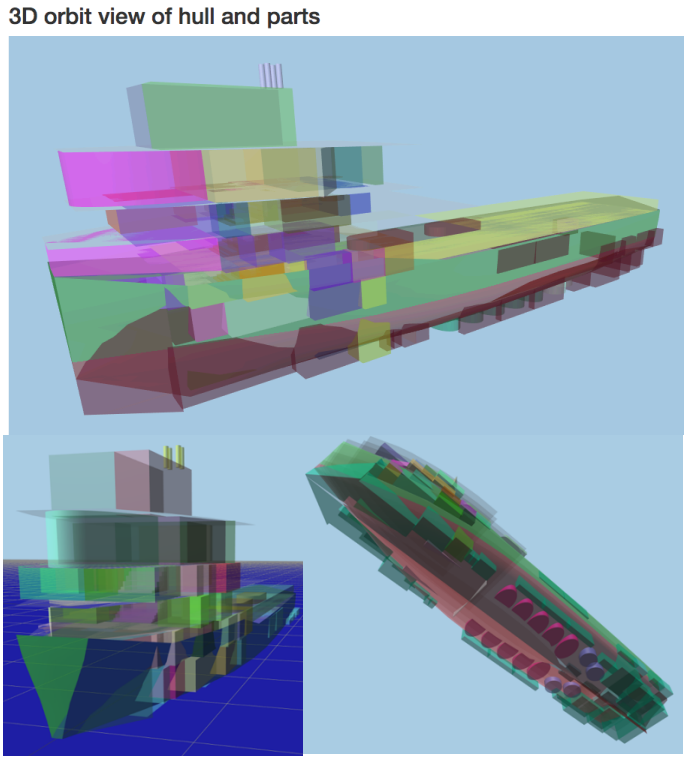


Figure 7 – PSV composed of 106 derived objects

5 VESSEL.JS FOR DESIGN AND SIMULATION

5.1 *Creating a design space*

Having one ship available is the starting point for more advanced analysis. A design space, for instance, can be created based in a unique ship, only by altering it objects' properties.

Fonseca *et al.* (2018) uses this concept for design space evaluation. The authors use *Vessel.js* to modify the existing hull from Figure 7 in 441 other vessels, with a step of 1% in the scaling ratio, but keeping all designs with the same displaced volume. The algorithm fixes one of the main dimensions (for instance length) and vary breadth and depth to achieve the same volume of the original vessel with a different configuration. Weights of hull, decks and bulkheads are scaled with their areas. Lightweights of tanks and compartments are assumed to remain constant. As capacities of tanks and compartments scale with volume, they also remain constant.

This design space consists thus of 441 different *Ship{}* objects, each of them with arrangements similar to the original ship, but with unique hull forms and main dimensions, as well as automatic modifications in the general arrangements, with rules for modify tanks dimensions and other derived objects that can be scalable.

Four examples of these 441 designs are observed in Figure 8, with the *Length, Beam* and *Depth* slides used for real-time 3D plot of each solution.

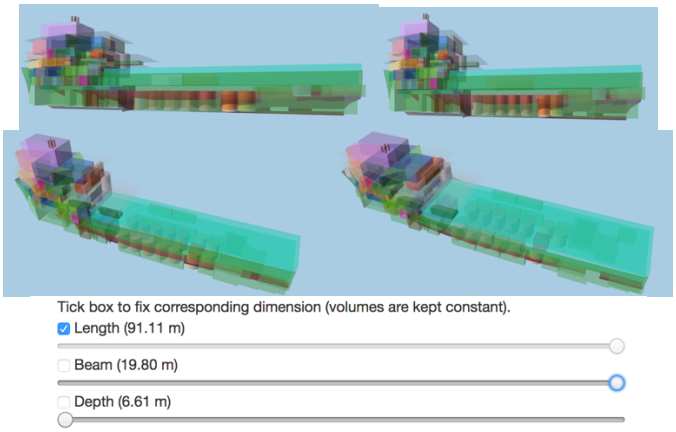


Figure 8 - Four different designs based on the PSV from Figure 7: each design has the same displaced volume, but different main dimensions and arrangement; a total of 441 designs with the same displacement was produced based in a unique PSV.

5.2 *Fuel-Consumption Simulation*

Simulating complex ship design performance is a key objective for the properly development of *Vessel.js*. These cases are not part per se of the *Vessel.js* library, but uses it for solving specific problems, and the objective is to have a gallery of validated processes that feed the PM examples required for a full virtual prototype experience (Figure 2). These PMs can then be copied, modified and extended according the designer needs. Few cases are now being developed for simulate process, here exemplified by combining resistance, seakeeping and fuel consumption simulation.

Basic fuel consumption can be calculated including engine and propeller information in the *Ship{}* object. The *Holtrop* method is able to provide the resistance of the hull for each speed and draft inputed, and information from propulsion efficiency and specific fuel consumption curves from the engine can be used to estimate the fuel consumed for a given draft in a given speed for a given period of time (Figure 9).

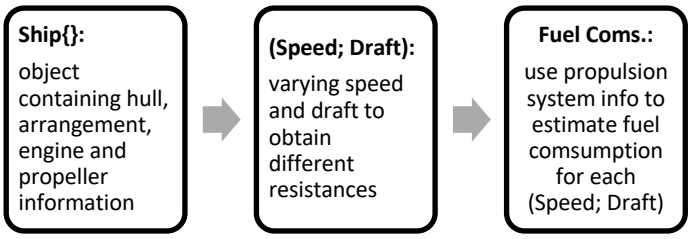


Figure 9 – Basic fuel consumption calculation based on a Ship{}, varying speed and draft to calculate resistance and consequently fuel consumption.

Using as basis the simple process presented in Figure 9, Fonseca *et al.* (2018) starts from the design space from Section 5.1 to evaluate the fuel consumption of 441 different designs, for the same path and under same wave condition. Besides *Holtrop*, the method also includes a wave resistance factor, and uses closed-form functions for basic seakeeping (Jensen *et al.*, 2004). As input, the user enters a preferred speed, and a maximum vertical acceleration is selected as constrain, as well a base time step, which will be used to calculate behaviour at different frequencies. For instance, in the input observed in Figure 10 the wave encounter is calculated each ten seconds, while fuel level is updated every minute and draft changes, due to fuel usage, every hour. Each of the 441 vessels is then simulated following a pre-defined path, with different wave heading angles and amplitudes for each of the legs of the path. When the vertical acceleration reaches the maximum threshold, the preferred speed is diminished until the threshold is reached. As output, total average speed and fuel consumption of the route is analysed for all designs.

Simulation Inputs

Parameter	Value	Parameter	Value
Preferred ship sailing speed, knots:	11.00	Wave time step, seconds:	10
Vertical acceleration threshold, m/s²:	3.00	Fuel level time step, seconds:	60
Planar path coordinates, nm:	[[0,0],[10,10],[0,0]]	Draft time step, seconds:	3600
Base time step, seconds:	1		

Figure 10 - Simulation inputs, including preferred speed, vertical acceleration limit, coordinates and time step data.

5.3 *Real-time Simulation*

Real-time visualization of time domain simulations is also a focus of the project, heavily influenced by past successful JS development, exemplified by Chaves and Gaspar (2016). Figure 11 illustrates their work, which consists of a web-based simulator, using closed-form expressions to estimate wave-induced motion for mono-hull vessels. These expressions require only vessel main dimensions and basic hull form coefficients, being especially relevant for conceptual design, where little information about the hull form is available. The approach allows the designer to vary those parameters, and quickly assess their influence on the wave-induced motion. Seakeeping closed-form expressions were implemented in JavaScript code, and transposed to time domain. In this way, parametric real-time simulations are provided by the application, together with a 3D visualization of vessel's motion in regular waves (heave, pitch and roll - Figure 11).

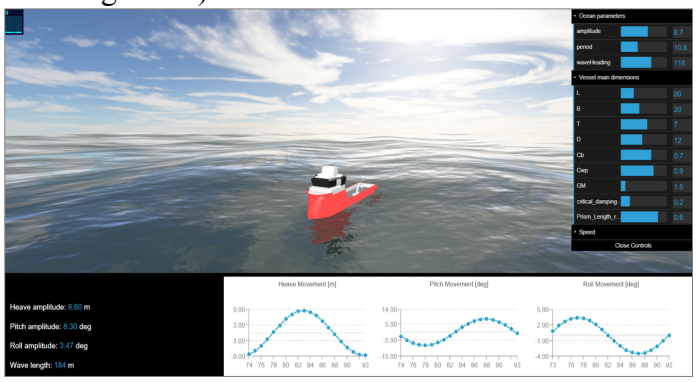


Figure 11 – Chaves and Gaspar (2016) simulator, developed in JS

The current case develops from the work from Chaves and Gaspar (2016) to standardize the input of the code as an object *Ship{}*, and adding the seakeeping methods in the *math* class. Real-time plot and 3D wave visualization is presented using the supporting libraries discussed in Section 3.4.

5.4 *Real-time Multibody Simulation*

The next challenge is to combine the simulation environment with the design space, in a multibody simulation. This research is yet under development, and the screenshot from Figure 12 presents the current stage, with a subset of the 441 cuboid-shaped bodies interacting in the same real-time environment. Each body is a different *Ship{}* object.

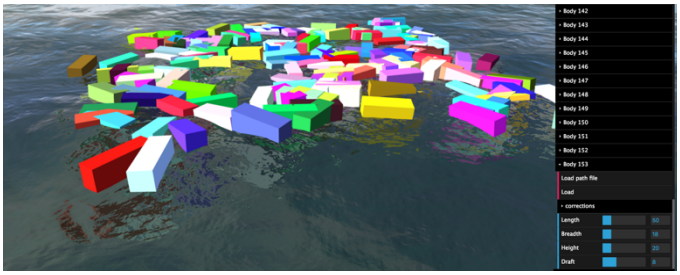


Figure 12 - Multibody real-time simulation: 441 design interacting in the same time-domain environment.

A magnitude of ±400 designs plotted in real-time in a modern computer did not presented lag for the real-time – each of the bodies movements in water were update within the 20-30 frame per seconds rate acceptable for human eyes, and waves seemed smooth. Therefore, to test the limits of current technology, a small research project was established to create the maximum of objects in the same environment using the current technology. The answer is a number a little less than 10^4 objects, each of them with unique geometry and material properties (Figure

13). This number can be raised if fewer unique instances are used, as well as computers with better graphical cards. On the other hand, the number is lower if we consider the sea and other details rather than cuboids for describing the ship.

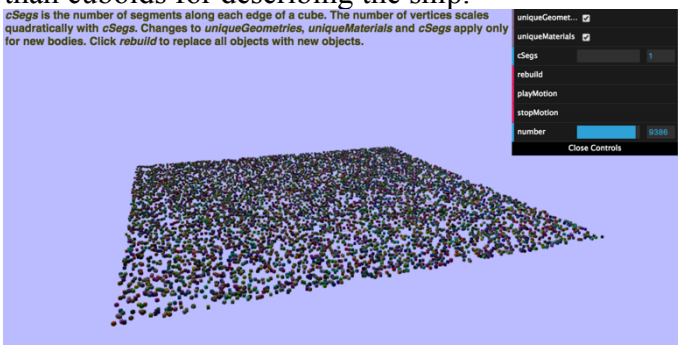


Figure 13 - Testing the limits of the real-time plotting of multibodies – a little less than 10^4 are a reality.

6 A CALL FOR DEVELOPMENT - EXTENDING THE LIBRARY AND CREATING A COMMUNITY

I close this paper with a call for my colleagues and students to consider developing future engineering analysis and simulation in JS. The examples here presented are a working in process, and much of the library structure and methods will be improved in the years to come. The main point defended in this paper is that technology is not a bottleneck for collaborative ship design, exemplified by the current fast-paced stage of online web-development, neither the speed of the computer processors and memory size, but rather how efficient ship design data is able to be transferred from books and experience to useful reusable models. Every naval architecture is able to follow Holtrop (1984) instructions to create their own Excel spreadsheet with hull resistance, but although a good exercise for students, this is a process that should be done once, and from there the function would be freely available to all, much similar to the functions available in Excel or Matlab – we rarely start from zero for a function that calculates the standard deviation of a series or a multiplication of matrices; why then are we yet re-typing and re-assembly 30-40 year old ship design functions?

Efficient handling of ship design data also faces the multifaceted aspect of its objects, with different taxonomies and hierarchies for solving different problems in each of the ship value chain phases. Finding a common standard *one size fits all* is practically impossible, and compromises must be done even in an open and adaptable object-oriented data formats, such as JSON.

As for the development of real engineering application in JS, I recognize the power of scientific applications of other languages, like Python, or long-term usability as robust Fortran codes; but combined with web elements, I defend that 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, and most of it is done by researchers connected somehow to the author. Commercial application is being tested by the ship design community in Norway (Gaspar, 2017), and collaborative research is done in cooperation the the Marine Group at UCL (Fonseca *et al.,* 2018; Piperakis *et al.*, 2018). The author is hopefully that in a matter of years much more will appear.

7 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. Moreover, *Vessel.js* is an open and collaborative library that could not be developed without the hard work from MSc students Elias Hasle, Ícaro Aragão, Alejandro Ivañez, Mateus Sant'ana and Olívia Chaves, and the given credits to each of their contribution is stated at the project page (http://vesseljs.org) – this paper would be possible without them.

8 REFERENCES

ANDREWS D. J., 2003. A Creative Approach to Ship Architecture. IJME No. 145 Vol. 3, September.

BOSTOCK, M.; V. OGIEVETSKY; J. HEER. 2011. D3: Data-Driven Documents, IEEE Trans. Visualization and Computer Graphics.

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

FEW, S. 2013. Now you see it, Analytics Press

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

FONSECA, Í.A.; GASPAR, H.M. 2015. An object-oriented approach for virtual prototyping in conceptual ship design, ECMS 2015, Varna.

FONSECA, I. A., GASPAR, H. M., RYAN, C., THOMAS, G. 2018. An Open and Collaborative Object-Oriented Taxonomy for Simulation of Marine Operations. 17th COMPIT, Pavone, Italy (accepted).

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.

GASPAR, H. M. 2018. Data-Driven Ship Design, 17th COMPIT, Pavone, Italy (accepted)

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

HE, B.; Y. WANG; W. SONG AND W. TANG. 2014. Design resource management for virtual prototyping in product collaborative design. Proc. IMechE, Part B: Journal of Engineering Manufacture, 1-17.

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

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

LEVANDER, K. 2011. System Based Ship Design Kompendium, Lecture Notes, NTNU, Trondheim

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

PARSONS, M. G. 2011. “Parametric Design”. In Ship Design and Construction vol. 1. SNAME.

PIPERAKIS, A., GASPAR, H.M PAWLING, R., ANDREWS, D. 2018. Designing Ships from the Inside Out - A Collaborative Object-Oriented Approach for the Design Building Block Method (unpublished)

GASPAR, H. M. 2017. JavaScript Applied to Maritime Design and Engineering 16th International Conference on Computer and IT Applications in the Maritime Industries, 2017.

ROH, M., KYU-YEUL LEE, K. “Computational Ship Design”, Springer Nature Singapore Pte Ltd. 2018, ISBN 978-981-10-4885-2.

WATSON, D. G. M., & GILFILLAN, A. W. (1976). Some ship design methods. Transactions of the Royal Institution of Naval Architects, 119, 279–324.