Ship Impact Model Version 1.0 – Instruction Booklet

1. How to Run

Start by downloading Python 3.4 as is on Dropbox, Anaconda installs Python, a integrated development environment, for editing and running files, Spyder, and the most frequently used packages.

The numpy package is heavily used for maths and the user interface (UI) uses tkiner and ttk.

Ensure all the program files are in the same folder.

Open “***inputdata.py***” and click run from Spyder to run the program.

Click a “preset” ship a the top KCS and KVLCC are represented by VLCC and Container Ship. This will populate the interface with “recommended values”, you can then change some of the characteristics as you wish.

1. Status

This version is completely missing a number of key functions, space has been made to allow for these to be added:

* Weather routing (Strathclyde).
* Fouling and engine degradation (Strathclyde).
* Equipment/weight breakdown (there are some simple estimates at the moment).
* Propeller model (this has been re-written in Python but needs testing – a simple estimate is used) (UCL). This is in work at the moment.
* Operating profile though this it is defined and linked in to the program via the file ***readinput.py***, the next step is running the program for the operational performance evaluation (UCL).
* Technology Interface and Waste Heat Recovery (UCL).

The UI also needs updating to reflect variables that have been added; the variables not currently in the interface are in ***main.py***.

There is an error with the GZ/righting moment calculator. It appears to be giving the wrong results.

There is also an error when changing between ships and for the hull generation of the oil tanker.

Some missing functionality is still in Matlab version of program.

1. Testing Strathclyde and Newcastle Models

There should be enough functionality here to test the link between the models. UCL has put dummy variables in each function so it will run, from now on each function is the owners responsibility to update, these are:

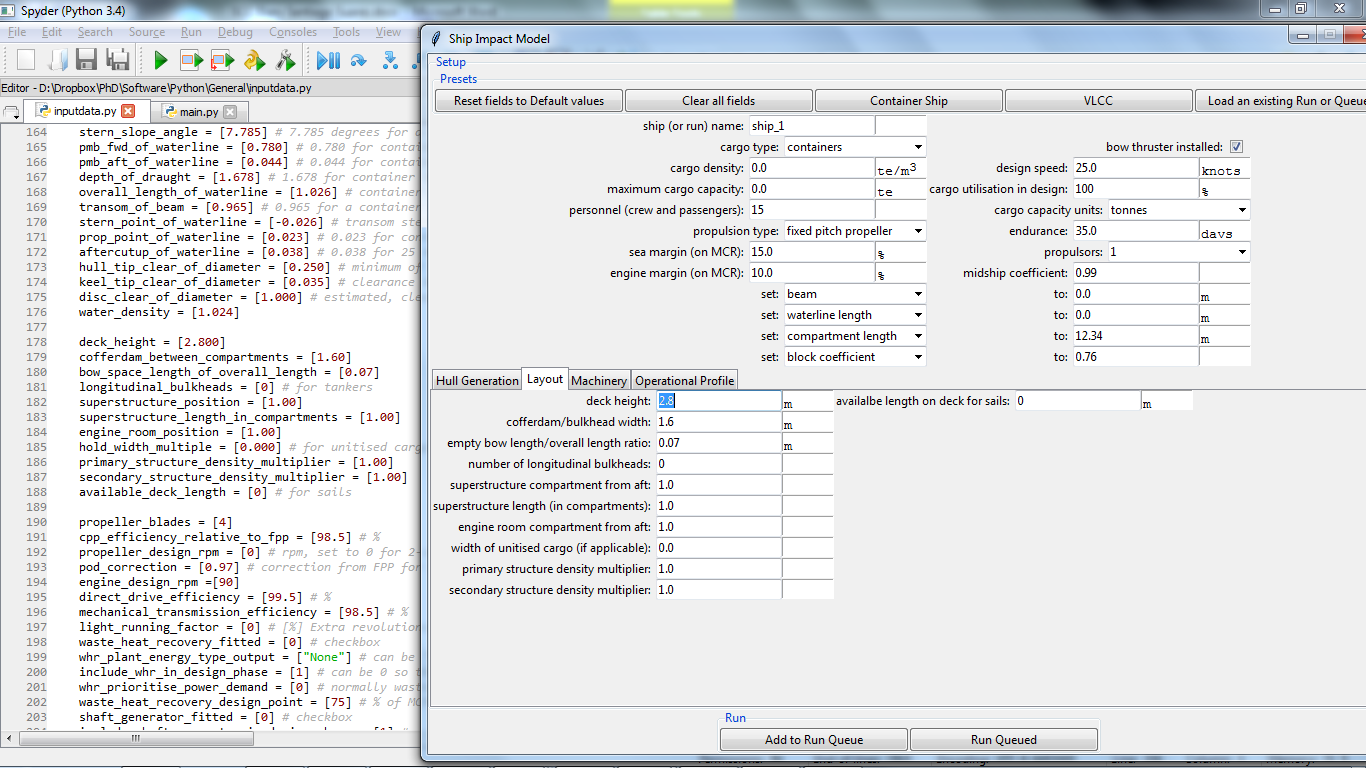
* Engine Model (Newcastle)
* Added Resistance Model (Strathclyde)
* Wind Model (Strathclyde)

1. How to add items to the user interface

Instances of variables have to be declared in several places in the UI, ***inputdata.py***. Therefore it is recommended that this is kept to a minimum where possible. The developer must use the latest files found in Github SCC/sim folder (https://github.com/Shipping-in-Changing-Climates).

The UI window is sized automatically by the code so in this regard the developer does not have to worry about the fit of the different variables. There will be instances where there is not more space to add variables – in version 1.0 this is only the case inside the tabs –, if this is the case then the developer should add a new tab where the variables can fit. It is important to have in mind that the tabs should group similar variables so the user can navigate through the tabs and find the variables intuitively and easily.

In the same manner, the code structure for inputdata.py reflects the same organization as in seen in the tabs by adding a blank space in between lines of code. The developer has to have this on mind when modifying the UI code in order to enable other developers to find variables easily throughout the different code sections.



In order to add a variable instance in the UI it is required to add 9 different lines of code inside inputdata.py, of which can be grouped as follows:

* Initial Values 🡪 5 lines of code
* Widgets 🡪 3 lines of code
* Interface Layout 🡪 1 line of code

To explain how the addition of new variables to the UI a simple example will be used throughout the different steps required. In the example the variable *endurance* is going to be added to the UI. For each variable that is added to the UI it is required another variable which must be called as the variable that is going to be added plus “\_entry” at the end. So in the case of *endurance*, the second variable will be called *endurance*\_entry .

* 1. *Initial Values*

In this section the default values for each variable and type of ship are declared.

The five lines of code needed are for class data and inside the action class the functions set\_default\_data, clear\_all\_fields, container\_ship and vlcc.

* + 1. The first line of code to be added is inside the data class: *endurance* = [35.00] 🡪 this is used to define the initial value.
    2. Inside the action class under the set\_default\_data function: data.*endurance*[0] = 35.00 🡪 This action will reset the values in the UI to its original values.
    3. Inside the action class under the clear\_all\_fields function: data.*endurance*[0] = 0.00 🡪 This action will reset the values in the UI to zero (in most of the cases).
    4. Inside the action class under the container\_ship function: data.*endurance*[0] = 35.00 🡪 This action will reset the values in the UI to a “preset” of values for a container ship .
    5. Inside the action class under the vlcc function: data.*endurance*[0] = 35.00 🡪 This action will reset the values in the UI to a “preset” of values for a vlcc ship.
  1. *Widgets*

After declaring the values for the variable (i.e. endurance), the code interacts with the UI to send, at the request of the user, the different values. This task is done by the widgets.

* + 1. Inside the function set\_values\_in\_entry\_widgets the next line of code must be written: data.*endurance\_entry*.input.set(data.*endurance*[0]).
    2. The next line of coding is inside add\_to\_queue function where there is an “if” conditional, so this means that it is required to have two lines of coding; one in case the condition is true and another for the false clause. If the variable run is equal to 0, then the following has to be written: rundata.*endurance*[run] = *endurance\_entry*.input.get().
    3. In case run is other number than 0, then the following line of code has to be added in its corresponding category after the else: rundata.*endurance*.append(*endurance\_entry*.input.get())
  1. *Interface Layout*

Finally there are functions to refer to the type of object in the UI. In this example the line of code has to be the following:

*endurance\_entry* = labelentry(setupframeright, "endurance: ", DoubleVar(), data.*endurance*[run], "days ")

In a generic way the code line is 🡪 variable\_entry = type\_of\_representation(UI\_location, “Text for the variable”, variable\_type, data.variable[run], “units”)

Inside ***inputdata.py*** there are three different type\_of\_representation which are:

* labelentry: single variable value, for example 35.
* labellist: displays a range of different values for the variable, this generally are recorded as text and have to have the structure given: “{cargo} {oil}”.
* labelcheck: works like an on/off switch.

Is important to mention that each type of type\_of\_representation needs different inputs and structure. It is recommended for the developer to follow the same structure of each type inside the ***inputdata.py***.

In this particular case the *endurance* variable will be displayed in the upper box of the UI, this box is split in the middle by an invisible vertical line creating two different boxes called setupframeright and setupframeleft which will be the values accepted for UI\_location . Each tab in the UI will have a different name for the UI\_location. The developer can find these names at the beginning of each box or tab created in ***inputdata.py***.

The text for the variables has to be inside quotes and has to be clear enough so the user can understand for what is the space given in the UI.

variable\_type is of the type double (DoubleVar()) or string (i.e. StringVar()), and depends on what is going to be displayed. labelentry tends to be double while labellist uses string.

Units have to be inside quotes if required. If the last letter or number of the unit has to be superscript then no space has to be left between the letter/number and the quote:

* “m3 “ 🡪 m3
* “m3” 🡪 m3

.

Finally, it is recommended to the developer to do a run test to check for errors and see if the location and aspect of the new variable in the UI is correct.

1. Reading comments and testing

HERE IN WORK is used to denote incomplete things. In order to test variables you can use the print() command, to print() them to the output window, similar to disp() in Matlab.

1. Design Process

The ***main.py*** contains the main program that calls functions for different calculations.

Currently the design process is carried out in the following order:

Initial Displacement Estimate

↓

Hull Generator

↓

KG Estimate (based on payload volume/weight fraction)

↓

Still water resistance model

↓

Weather Routing

↓

Added Resistance

↓

Propeller Efficiency Estimate

↓

Heel Angle Estimate

↓

Wind Assist

↓

Technology Interface

↓

Marine Systems and Engine

↓

Waste Heat Recovery

↓

Shaft Generator

↓

Marine Systems and Engine

↓

2nd Loop

Equipment and Structure

↓

Hull Generation

↓

Still Water Resistance Model