Preface

ModelUI provides a generic interface for modelling applications that produce some combination of graphical and/or time series outputs. The purpose of this user interface (UI) is to enable the rapid prototyping of models by allowing the model developer to focus on the model, rather than the functional or operational needs of the software package itself. To this end, the UI provides a standard interface with drop-down menus, tools to open and close files, keep track of model runs, provide a rapid means to implement model set-up and data import and export, derivation of new variables, and some basic plotting and statistical tools. The core ModelUI package includes several example models to illustrate how to create and modify the UI for different applications. These include:

VerticalProfile to illustrates various options to set up a model in ModelUI.

SimpleTide to illustrate some alternative setup options, whilst retaining the default UI.

Diffusion2D to illustrate how to handle time plus 2 or 3 space dimensions.

Requirements

The model is written in MatlabTM and provided as Open Source code (issued under a GNU General Public License) and runs under v2016b or later. ModelUI uses the muitoolbox and dstoolbox.

Resources

The ModelUI App and two toolboxes (muitoolbox and dstoolbox) can be downloaded from [www.coastalsea.uk](http://www.coastalsea.uk).

Bibliography

The models used to illustrate the UI are based on standard published methods, including:

Prandle, D., 1982. The vertical structure of tidal currents and other oscillatory flows. Continental Shelf Research, 1(2), 191-207.

See also the manuals for:

CoastalTools - <http://www.coastalsea.uk/download-page/coastaltools-2/coastaltools-manual/>

Acknowledgements

The Diffusion model is based on the code to solve the 2-D diffusion equation developed by Suraj Shanka, Copyright (c) 2012 and made available via the Matlab TM Exchange Forum.

Testing undertaken by Tian Qi.

Revision history

|  |  |  |
| --- | --- | --- |
| Version | Date | Changes |
| 3.0 | May 2021 | ModelUI packaged as a Matlab App and migrated to use muitoolbox and dstoolbox. |
| 2.1 | Dec. 2020 | Help text added to function. Various bug fixes. Package core suite as a toolbox. |
| 2.0 | July 2019 | Restructured UI, Data and Model classes to make Data and Models independent of class handles defined for a specific UI. This allows Data and Model classes to easily be used in any UI, or for multiple models to be included within a single UI. Unfortunately, this means that Classes developed for version 1.1 need to be updated (a relatively minor task) because they are no longer compatible.  All imported data sets and model outputs now inherit either TSDataSet for timeseries data and DSDataSet for table data. These super classes in turn inherit the generic functions in DataSet. This allows data sets with different formats to be included in the same class (e.g. imported wave data with different variables and formats).  Added descriptive and timeseries statistics class (DataStats) and range of functions (in MUIfunctions) so that Statistical analysis can be included in the UI.  Added panel to tabs in DataGUIinterface to improve behaviour  Improved behaviour when running multiple UIs/Models |
| 1.1 | June 2018 | Minor corrections to improve cross-platform compatibility  Fixed bugs when setting data ranges in Data GUIs and selecting data from a pop-up list.  Improved plotting option selection for xyz data with no time  Scenario selection added to control of Plot tab |
| 1.0 | Mar 2018 | Full release with some updates to code but no changes to model structure or requirements. |
| 0.1 | Jan 2018 | Preliminary release via www.coastalsea.uk |

Contents

[1 Introduction 1](#_Toc58851109)

[2 Getting started 2](#_Toc58851110)

[2.1 Configuration 2](#_Toc58851111)

[2.2 Model Set-up 2](#_Toc58851112)

[3 Standard Interface 3](#_Toc58851113)

[3.1 File 3](#_Toc58851114)

[3.2 Tools 3](#_Toc58851115)

[3.3 Project 3](#_Toc58851116)

[3.4 Setup 4](#_Toc58851117)

[3.5 Run 4](#_Toc58851118)

[3.6 Plot 8](#_Toc58851119)

[3.7 Tabs 9](#_Toc58851120)

[4 Demonstration models 10](#_Toc58851121)

[4.1 Vertical Tidal Current Profile 10](#_Toc58851122)

[4.1.1 Workflow to Run Model 11](#_Toc58851123)

[4.1.2 Plotting results 11](#_Toc58851124)

[4.2 Simple tide 11](#_Toc58851125)

[4.3 Breach model 12](#_Toc58851126)

[4.4 InshoreWaves model 12](#_Toc58851127)

[4.5 Diffusion model 12](#_Toc58851128)

[4.6 Derive Output 13](#_Toc58851129)

[5 Adding a new model to ModelUI 17](#_Toc58851130)

[5.1 Data input class 17](#_Toc58851131)

[5.2 Model function file 18](#_Toc58851132)

[5.2.1 Model output formats 19](#_Toc58851133)

[5.3 Model specification in ModelUI 20](#_Toc58851134)

[5.4 Models provided to run within ModelUI 21](#_Toc58851135)

[5.5 Alternative method to set up a Model using the default interface 21](#_Toc58851136)

[6 Implementing a new bespoke model 24](#_Toc58851137)

[6.1 Using ModelUI for a different model 24](#_Toc58851138)

[6.1.1 The User Interface 25](#_Toc58851139)

[6.1.2 Data Input and Model classes 27](#_Toc58851140)

[6.2 Data input 31](#_Toc58851141)

[6.3 UI customisation 32](#_Toc58851142)

[6.4 Timeseries data 35](#_Toc58851143)

[6.5 Categorical data 37](#_Toc58851144)

[6.5.1 Adding a new table 37](#_Toc58851145)

[6.5.2 Adding a new table class 37](#_Toc58851146)

[6.6 Multi-dimensional variables 38](#_Toc58851147)

[6.7 Altering the Plotting UI 39](#_Toc58851148)

[6.8 Additional programming notes 39](#_Toc58851149)

[6.8.1 Data assignment 39](#_Toc58851150)

[6.8.2 Defining the model output properties 41](#_Toc58851151)

[6.8.3 Checking for a valid model 41](#_Toc58851152)

[6.8.4 Data structure 42](#_Toc58851153)

[6.8.5 Selections on tabs in DataGuiInterface 43](#_Toc58851154)

[6.8.6 Permissible ways of adding data to a class handle or property 44](#_Toc58851155)

[6.8.7 Data retrieval in classes that use DataGuiInterface 45](#_Toc58851156)

[6.8.8 Supressing the Scenario prompt 45](#_Toc58851157)

[6.8.9 Input file formats 45](#_Toc58851158)

[7 Program Structure 47](#_Toc58851159)

[8 Bibliography 48](#_Toc58851160)

[Appendix A – DataGuiInterface function calls 49](#_Toc58851161)

[Appendix B – Installation using zip file 50](#_Toc58851162)

# Introduction

Whether prototyping a new method or writing a new application, dealing with the coding infrastructure needed to handle the user interface, data input/output and plotting the results can be time consuming. ModelUI aims to speed up the process and provide users with a standard UI that can be rapidly adapted for new applications. This manual explains the functionality of the standard UI and how it can be used as the UI for other models. There are four sample models to illustrate different ways of using ModelUI. Within ModelUI there is a simple model for vertical velocity profiles. The SimpleTide model illustrates how to replace this model without changing the UI. The Diffusion2D model illustrates how to handle multi-dimensional variables. These models are briefly summarised in Section 4 and the detail of each model is given in their respective manuals.

The models supplied with ModelUI are used to illustrate how to implement some basic used of ModelUI and the muitoolbox. There are two ways of doing this:

1. using the ModelUI interface ‘as is’ and adding a new model to the interface. The steps involved to do this are explained in Section 5;
2. implementing a bespoke interface to provide the required functionality. How to do this is explained in Section 6.

A simple model may only require the addition of one new class to implement. However much more sophisticated applications are also possible. The ASMITA and CoastalTools applications use the same model interface. They also provide a resource for many data handling methods that may be of use for other applications. These include some statistical methods, as well as a variety of plotting options.

# Getting started

## Configuration

ModelUI is available as a toolbox (file: MUImodels\_toolbox.zip) or as a set of folders (file: ModelUIcore.zip). The source code is accessible for both. The toolbox is installed as an Add-On in Matlab™ and all the folder paths are initialised upon installation and the location of the code, documentation and example files is also handled by Matlab. The folder option must be installed manually and folder paths set when using a particular model. See Appendix B – Installation using zip file further details.

The toolbox zip file contains the toolbox installation file, MUImodels.mltbx, a copy of the ModelUI licence and user manual. The toolbox is installed in Matlab using the Manage Add-Ons option from the Add-Ons icon on the Home tab. Once installed, the demonstration model can be run from the Command Window using:

>> ModelUI;

Documentation can be viewed from the Supplementary Software in the Matlab documentation.

## Model Set-up

*File>New* to create a new project space.

*Setup>Input Data>Model Data*

The UI requests data for the model variables. Once added the current set of variables can be viewed using the *Inputs* tab.

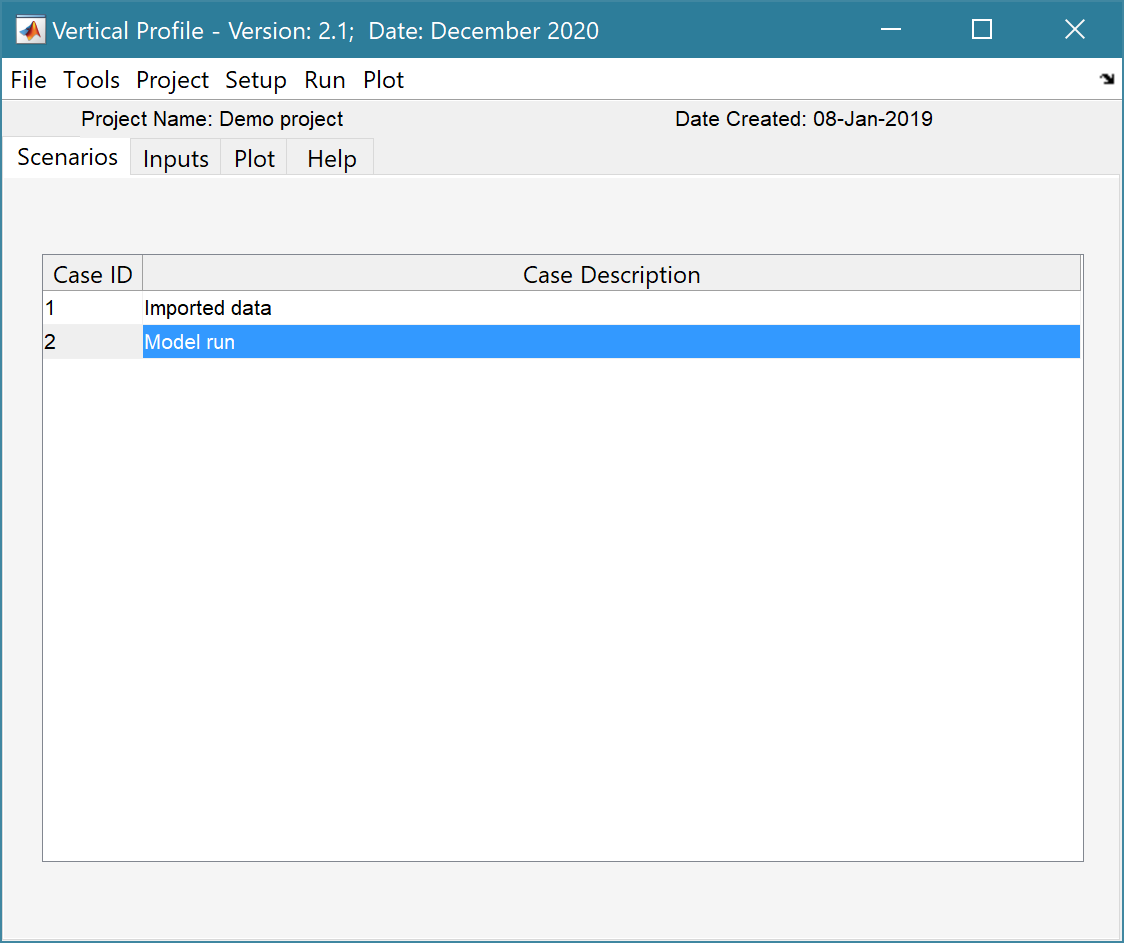
*Run> Run model*

When the run has completed the user is prompted to provide a description of the model run (scenario).

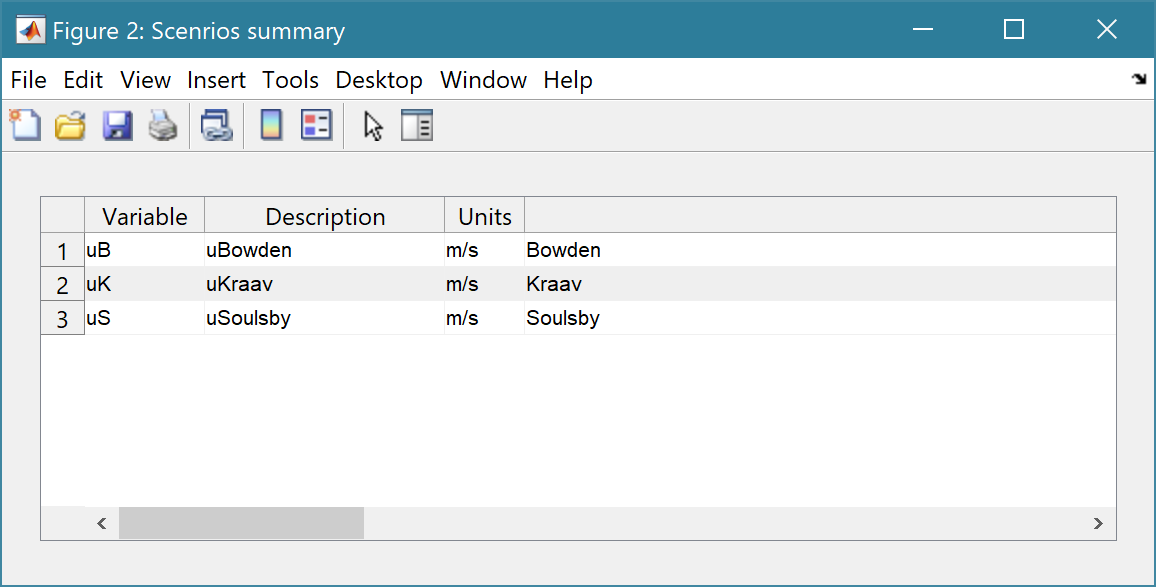
The run is listed on the *Cases* tab and the tidal elevations for the most recent run can be viewed on the *Plot* tab.

*Plot>Plot menu*

The results from a run can be selected and plotted. By using the Add button additional model runs can be included on the plot, allowing different Cases to be compared.

 Default User Interface

Left click mouse on a Case to see the associated meta-data (as shown below)



# Standard Interface

The GUI comprises a series of drop down menus that provide access to a number of commonly used functions such as file handling, management of run scenarios, model setup, running and plotting of the results. In addition, Tabs are used to display set-up information of the Cases that have been run. In this manual text in *Red italic* refers to drop down menus and text in *Green italic* refers to Tab titles.

## File

*File>New*: clears any existing model (prompting to save if not already saved) and a popup dialog box prompts for Project name and Date (default is current date).

*File>Open*: existing models are saved as \*.mat files. User selects a model from dialog box.

*File>Save*: save a file that has already been saved.

*File>Save as*: save a file with a new or different name.

*File>Exit*: exit the program. The close window button has the same effect.

## Tools

*Tools>Refresh*: updates *Cases* tab.

*Tools>Clear all>Project*: deletes the current project, including setup parameters and all Cases.

*Tools>Clear all>Figures*: deletes all results plot figures (useful if a large number of plots have been produced).

*Tools>Clear all>Case*: deletes all cases listed on the  *Cases* tab but does not affect the model setup.

## Project

*Project>Project Info*: edit the Project name and Date

*Project>Cases>Edit Description*: select a scenario description to edit

*Project>Cases>Edit Data Set*: edit a data set. Initialises a data selection UI to define the record to be edited and then lists the variable in a table so that values can be edited. The user can also limit the data set retrieved based on the variable range and the independent variable (X) or time. This can be useful in making specific edits (eg all values over a threshold or values within a date range).

*Project>Cases>Save*: user selects the scenario to be saved from a list box of Cases and the user is then prompted to name the file. The full details of the model setup and the results are written to an Excel spreadsheet.

*Project>Cases>Delete*: user selects the scenario to be deleted from a list box of Cases and results are then deleted (model setup is not changed).

*Project>Cases>Reload*: select a previous model run and reload the input values as the current input settings.

*Project>Cases>View settings*: displays a table of the model input parameters used for a selected model run.

*Project>Export/Import>Export*: saves a timeseries data set to a Matlab binary ‘mat’ file.

*Project> Export/Import>Import*: loads data from a Matlab binary ‘mat’ file. Only works for data sets saved using Export. These two functions can be used to move data sets between projects or models.

## Setup

The setup menu provides a series of menus to enable different components of the model to be defined.

*Setup>Input Data>Model Data*: Allows the user to enter and edit model parameters (generates a UI for the exposed Properties of the Model class).

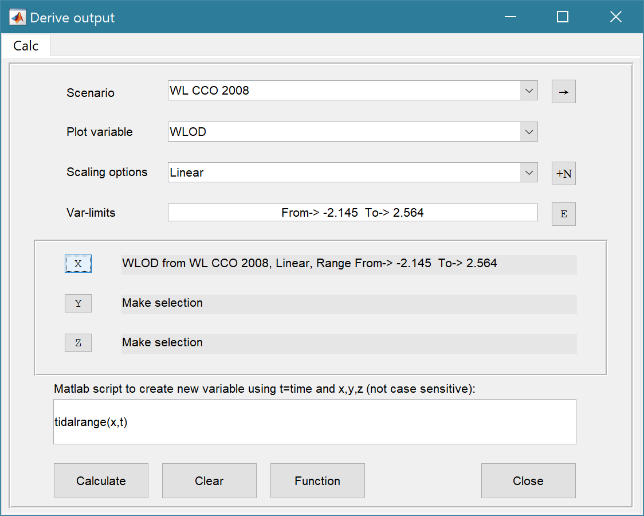
*Setup>Input Data>Imported Data*: Select one or more files to load. Once added the current set of variables can be viewed using the *Inputs* tab. When the data has been loaded, the user is prompted to provide a description of the data set (scenario) and is listed on the *Cases* tab. The source file(s) area also listed on the *Inputs* tab.

*Setup>Input Data>Model Constants*: Constants, such as water and sediment density can be defined to be used in different models without having to redefine them. Generally, the default values are appropriate, but these can be adjusted and saved with the project if required.

*Setup>Model Specs*: This option allows model specifications to be added, edited or deleted. This is part of the set-up for a new model and is explained in more detail in Section 5.3.

## Run

*Run> Run Model*: runs model, prompts for scenario description which is added to the listing on the *Cases* tab.

*Run> Derive Output*: data that has been added (either as data or modelled values) can be used to derive new variables. The UI allows the user to select data and use a chosen selection of data/variable/range to define either a Variable, XYZ, or Time. Each data set is sampled for the defined data range. If the data set being sampled includes NaNs the default is for these to be included (button to right of Var-limits is set to ‘+N’). To exclude NaNs press the button so that it displays ‘-N’.

The selection is assigned by clicking one of the X, Y or Z buttons. The user is prompted to assign a Variable, XYZ, or Time (the options available varies with the type of variable selected). An equation is then defined in the text box below using the x, y, z or t variables[[1]](#footnote-1). Based on the user selection the routine applies the defined variable ranges to derive a new variable.

If the result from the equation is single valued, this is displayed in a dialogue box. If the result is a vector or array, the user is prompted to name the variable and provide additional descriptive information before the variable is saved. There are three options:

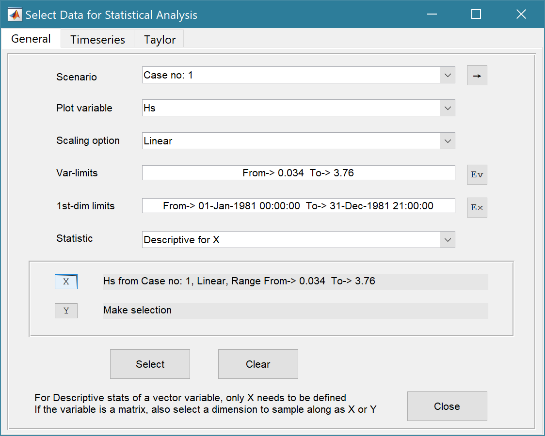
1. Create a New Definition. This is the only option the first time the UI is used to derive a new variable. The user is prompted for all the information needed to define an instance of a new data class.
2. Create a New Case. This uses an existing Derived class instance and adds a new record to the selected instance. This can be defined as a new Case and is displayed on the *Cases* tab but it uses the existing Derived class instance variable definitions (i.e. ResDef).
3. Add to an Existing Case. This adds the variable to an existing Case by adding a new variable to the tscollection or table. The user is prompted to define the new variable, and this is used to extend the existing Derived class instance variable definitions (i.e. ResDef). This can be inspected and edited using *Plot>Output Definitions*.

Some further details on using this option are provided in Section 4.6 and further implementation details are given in Section 6.8.6.

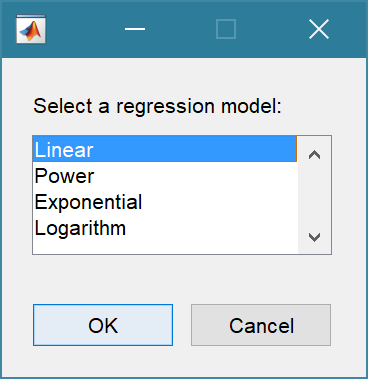
*Run> Statistics*: several statistical analysis options have been included within the Statistical Analysis GUI. Currently three *tab* options are defined but the number of these used in an application is defined in the UI setup file (see Section 5.5). The tabs currently defined are for General statistics, Timeseries statistics and for model comparisons using a Taylor Plot.

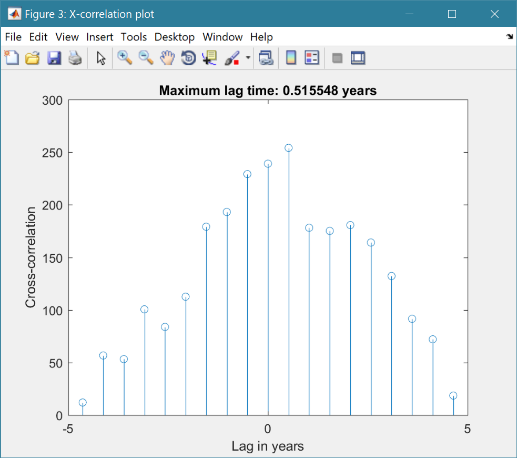
***General tab***

The General tab allows the user to apply the following statistics to data loaded in ModelUI:

1. **Descriptive for X**: general statistics of a variable (mean, standard deviation, minimum, maximum, sum and linear regression fit parameters). Only X needs to be defined. The variable and default (1st) dimension can be adjusted. However, this should be done BEFORE assigning the variable to the X button. If the variable being sued is a multi-dimensional matrix (>2D), the user is prompted to define the range or each additional dimension, or select a value at which to sample. The function can return statistics for a vector or a 2D array.

The results are tabulated in a new window and can be copied to the clipboard for use in other applications.

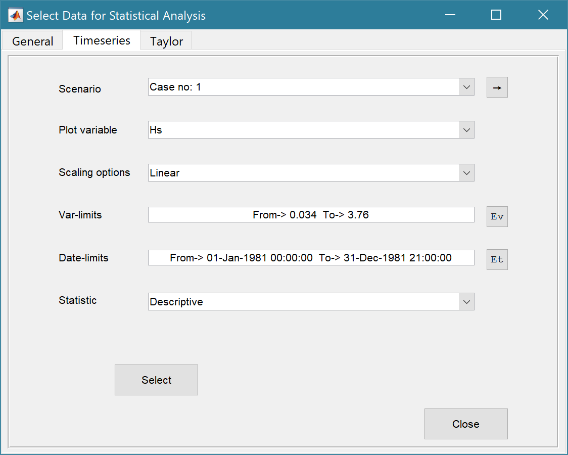
1. **Regression**: generates a regresion plot of the independent variable, Y, against the depended variable, X. For time series data, the default data range is the maximum period of overlap of the two records. For other data types the two variables must have the same number of data points. After pressing the Select button, the user is prompted to select the type of model to be used for the regression. The results are output as a plot with details of the regression fit in the plot title.

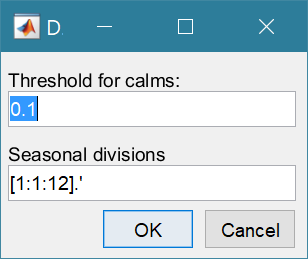


1. **Cross**-**correlation**: generates a cross-corrleation plot of the reference variable, X, and the lagged variable, X (uses the Matlab ‘xcorr’ function). For time series data, the default data range is the maximum period of overlap of the two records. For other data types the two variables must have the same number of data points. This produces a plot of the cross-correlation as a function of the lag in units selected by the user.

***Timeseries tab***

The Timeseries tab allows the user to apply the following statistics to data loaded in ModelUI:

1. **Descriptive**: general statistics of a variable (mean, standard deviation, minimum, maximum, sum and linear regression fit parameters). The results are tabulated in a new window and can be copied to the clipboard for use in other applications.

Various ‘seasonal’ sub-divisions can be defined. The required option is selected from the table in the UI, by selecting a Syntax cell and then closing the UI.

The next UI prompts for a threshold for calms (values below threshold are deemed to be “calm” conditions) and allows the selected ‘seasonal ‘divisions to be changed (if the desired option is not in the default list), or edited. The divisions can be expressed in several ways, as detailed below:

|  |  |
| --- | --- |
| **Script** | **Result** |
| 1 | Descriptive statistics for the full-time series |
| [1:1:12].’ | Descriptive statistics for the full-time series and monthly values (the .’ creates a column vector). |
| [12,1,2; 3,4,5; 6,7,8; 9,10,11] | Descriptive statistics for the full-time series and seasons based on groupings – Dec-Feb, Mar-May, Jun-Aug, Sep-Nov shown. |

1. **Peaks**: generates a new timeseries of peaks over a defined threshold. There are three methods that can be selected:

1 = all peaks above the threshold;

2 = the peak value within each up-down crossing of threshold; and

3 = peaks that have a separation of at least ‘*tint*’ hours.

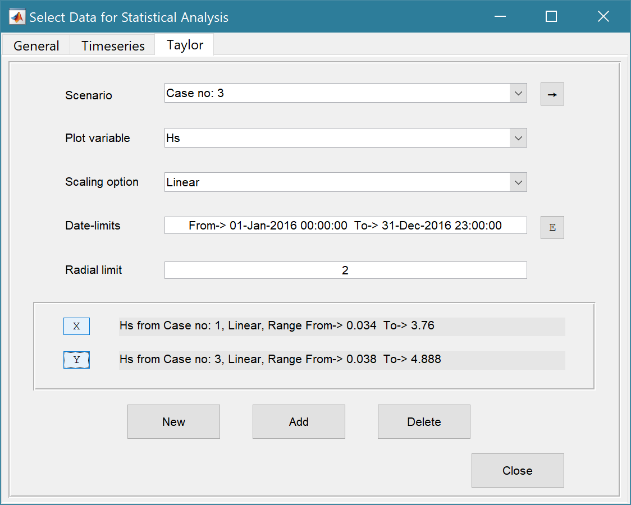
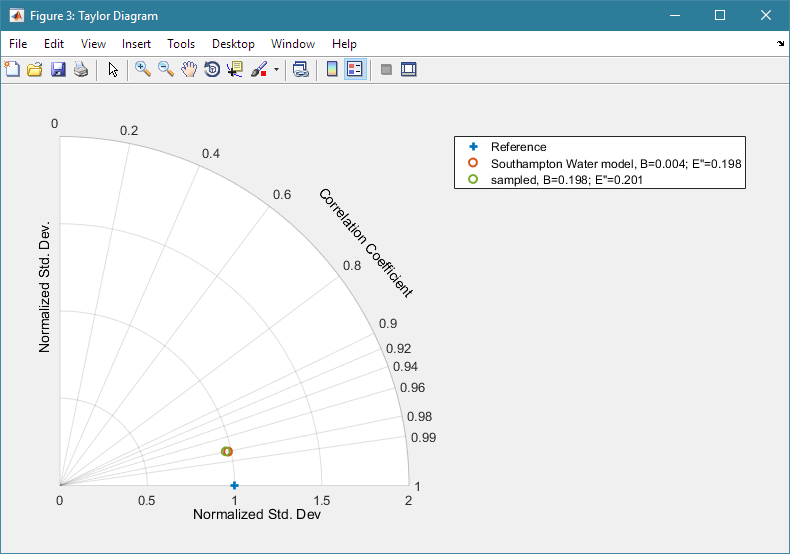
For option 3, the separation between peaks (‘*tint*’) is also be defined in the pop-up gui. This can be used to try and ensure that peaks are independent. The peaks are marked on a plot with the defined threshold. If rejected, new values can be defined. If accepted a new timeseries is added. This has the class of the Data Type that was used as the source timeseries but is not appended to that timeseries because the date/times are a subset of the source.

1. **Clusters**: The selection process is similar to peaks, where the user defines a threshold, selection method and time between peaks (for method 3). In addition, the cluster interval is defined in days. This is the period of time separating two peaks for them to be no longer considered part of a cluster (e.g. if a sequence of storms occurs every few days they will form a cluster. If there is then a gap of, say, 31 days to the next storm, with a cluster time interval of 30 days this would be considered as part of the next cluster). Once a selection has been made, a plot is generated that shows the peaks for each cluster with a different symbol. The user can either choose a different definition, or accept the definition. Once accepted, the results are added as a new timeseries, with the class of the Data Type that was used as the source timeseries. Two values are stored at the time of each peak in the clusters: the magnitude of the peak; and the number of the cluster to which it belongs (numbered sequentially from the start). This allows the data for individual clusters to be retrieved, if required.
2. **Extremes**: The selection process is similar to peaks, where the user defines a threshold, selection method and time between peaks (for method 3). A figure is generated with two plots. The left-hand plot shows the peaks for the defined threshold and the right hand plots shows the mean excess above the threshold (circles), the 95% confidence interval (dotted red lines) and the number of peaks (vertical bars + right hand axis) as a function of threshold. This plot can be used to help identify a suitable threshold for the peak-over-threshold extremes analysis method. The user can either choose a different definition, or accept the definition. Once accepted, the user is prompted to select a plot type. Options are: None; Type 1 – a single return period plot; Type 2 – a composite plot showing the probability, quantile, return period and density plots. See Coles (2001) for further details of the method used and the background to these plots. The results are tabulated on the *Stats/Extremes* tab and can be copied to the clipboard for use in other applications.
3. **Poisson Stats**: user is prompted to select a threshold, method and peak separation (see Peaks above) and the function generates a plot of the peak magnitude, time between peaks (interarrival time) and the duration above the threshold for each peak. The plot shows a histogram of each variable and the exponential pdf derived from the data, along with the value for the fit.
4. **User**: calls UserStats.m function, where the user can define a workflow, accessing data and functions already provided by ModelUI. The sample code illustrates the work flow to produce a clusters plot. Some code in the header (commented out) shows how to get a time series using the handles passed to the function (obj and mobj) rather than the timeseries that is passed (ts). This code would get the same timeseries as the one passed to the function. However, by modifying the ‘options’ variable it is possible to access other timeseries variables.

***Taylor tab***

The Taylor tab allows the user to create a Taylor Plot using the data loaded in ModelUI:

A Reference time series and a Test time series are selected as X and Y respectively. The timeseries are clipped to a time-period that is common to both, or any user defined interval that lies within this clipped period. The statistics (mean, standard deviation, correlation coefficient and centred root mean square error) are computed, normalized using the reference standard deviation and plotted on a polar Taylor diagram (Taylor, 2001).



The user is also prompted to select whether to include the calculation of a skill score (Yes/No). As further points are added to the plot, this selection remains unchanged (i.e. the skill score is or is not included). To reset the option it is necessary to close and reopen the Statistics UI.

## Plot

*Plot>Plot menu*: initialises the Plot gui to select variables and produce several types of plot. The user selects the Scenario, the Variable to used and the Scaling option from a series of drop down lists. There are then buttons to create a New figure or Add or Delete variables from an existing figure.

|  |  |
| --- | --- |
|  | **X plot**  Assumes that the first variable (or time in timeseries data) is the X variable.  > Select a Scenario and Variable to plot.  The Y variable can be scaled (log, normalised, etc) and the range to be plotted can be adjusted from the minimum and maximum values shown in the Y-limits and X-limits boxes.  > Select plot type (line, bar, scatter, stem, etc)  Buttons:  → : updates the list of Cases  Y : swaps the X and Y axes  + : switches between cartesian and polar plot type |
|  | **XY plot**  > Select Scenario, Variable, Scaling and Limits  > Select plot type  > Assign selection to X and Y by clicking button. Selection is displayed in the text box.  > Repeat process for additional variables. |
|  | **XYZ plot**  > Select Scenario, Variable, Scaling and Limits  > Assign selection to X, Y or Z by clicking button. Selection is displayed in the text box.  > Repeat process for additional variables.  If only X and Y are assigned, then a 2D line plot is produced. Add and Delete can be used for 2D plots only.  For X, Y, Z selections, a 3D surface contour plot is produced. To produce a new plot, use the Clear button to remove the previous selection.  Buttons: see X plot for functions  Unless phase or direction are the X variable, Polar plots are best done using the XYZ tab. Selecting a 2D polar plot includes the option to produce a rose or polar plot. |
|  | **Animate plot (only used in Diffusion2D)**  Select Scenario, Variable, Scaling and Variable Limits  Apply and additional limits required for Time and X, Y and Z. If the data is T-XYZ then the user is prompted to select a 2D plane to use in the animation: |

For all plot types, when the data has more dimensions than the plot or animation the user is prompted to sub-select from the data (by selecting sampling values for the dimensions that are not being used).

*Plot>Output Definitions*: provides the ability to edit the Output Definitions used to define output variable names and labels for each model or data set. This edits the model or data set class property ResDef within the project but does not change the class definitions. To make permanent changes you need to edit the class definition of the property ResDef, which is assigned in the class constructor.

## Tabs

To examine what has been set-up the Tabs provide a summary of what is currently defined. Note: these update when clicked on using a mouse and the values cannot be edited from the Tabs.

*Cases*: lists the cases that have been run with a case id and description. Clicking on a row generates a table with details of the variables for the case and any associated metadata (e.g. the variables used for Derived Output – see Section 3.5). For table data sets, a summary is also printed in the Command Window.

*Inputs*: tabulates the system properties that have been set (display only).

*Plot*: plots the vertical profile for the most recent scenario (display only).

*Help*: an abridged version of the model workflow guide.

# Demonstration models

To illustrate the use of the interface for different types of output (graphical and time series), four sample models are provided. The first model is provided as the demonstrator model within ModelUI and computes the vertical tidal current profile. The second model generates a simple tidal curve, the third generates “design” sections for the breach of a seawall (as used in managed realignment projects), the forth uses an offshore wave timeseries to compute the inshore wave conditions at the edge of the surf zone and final model generates an animation of 2D diffusion. These models then illustrate the following types of change to the core functionality of ModelUI:

SimpleTide – emulates tidal elevation and velocity. There are two versions:

1. Runs within ModelUI, using a model specific class for data input and calling a function to implement the model;
2. Runs as a stand-alone model. This version replaces the Model class in ModelUI with a model specific class and uses the same data input class as (a).

MRBreach – has class, MRBreach that inherits ModelUI and modifies the menus and tabs of the GUI. This includes an additional tab to display some interim results for the site hypsometry, that can be adjusted prior to running the model. The other model specific classes illustrate how to replace an existing class (e.g for data input) and how to add additional classes (Hypsometry, BreachModel, and SiteData). In addition, MRBreachData and MRSiteData illustrate how to split the data input and display to make it more manageable.

InshoreWaves – has class InWave that inherits ModelUI and modifies the menus of the GUI. Data input is defined using InWaveData and In WaveSite. The main addition here is two classes to handle timeseries data by inheriting from DataSet. These are InWaveData to manage imported timeseries and InWaveModel to handle the timeseries created by the model. Both store the data in Matlab timeseries collections.

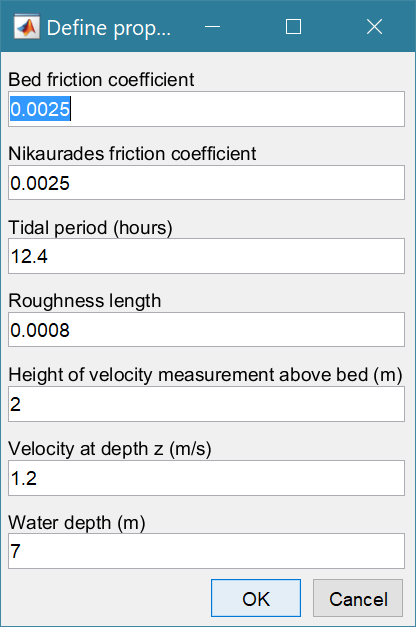
Diffusion2D – the adaptation of the ModeUI is similar to MRBreach and InshoreWaves. The main purpose of this model is to illustrate the use of Matlab tables to hold a multi-dimensional array that defines a variable at each time-step (eg 1, 2 or 3D arrays to represent a variable in x, y and z).

The models themselves are briefly summarised in the following sections, with further details given in the model specific manuals. How to modify ModelUI for a new application is explained in Section 6.

## Vertical Tidal Current Profile

The VerticalProfile sub-folder includes a function called verticalprofilemodel. This implements the approach outlined by Prandle (1982) to compute the variation of the vertical profile given measurements at one elevation. This implementation has the option to use several different eddy viscosity formulations when computing the profile.

### Workflow to Run Model

The following outlines the steps in a typical workflow to setup and run the vertical tidal current profile mode.:

*File>New* – create a new project (name and date)

*Setup>Input Data>Model Data* – define model parameters (see Prandle (1998) for details of the parameters required. To see the parameters that are currently set for the model use the *Inputs* tab.

*File>Save as* – save model setup to a \*.mat file.

*Run>Run model* – runs the model and prompts user for a description of the scenario. The results can be viewed on the *Plot* tab (Figure 1).

Completed ‘Cases’ are listed based on the user descriptions on the *Cases* tab.

### Plotting results

Results can be plotted using the plotting UI (*Plot>Plot Menu*). Eddy viscosity cases can be selected individually and plotted against profiles from other Cases.

Figure 1 – Vertical profiles from a model run plotted in UI Plot tab and comparing several Cases using the Plot Menu

|  |  |
| --- | --- |
|  |  |

## Simple tide

SimpleTide provides a simple representation of the diurnal-semidiurnal and spring-neap variations in the tide based on simple summation of M2, S2 and O1 contributions, scaled to the defined tidal range. Tidal currents are derived in a similar way and scaled to the defined tidal velocity amplitude. The model generates a time series of tidal amplitude, vertical and horizontal velocity based on defined amplitude and phases. The model is implemented in two ways, to illustrate different ways of adding models

1. By defining a new class, STData for data input and a model function file simpletidemodel. This runs within ModelUI and is similar to the implementation of the Vertical Profile model.
2. by defining three new classes: SimpleTide, STData and STModel. The SimpleTide class inherits from ModelUI and redirects menu options to STData and STModel. This illustrates a minimal adjustment to add a new model. The STData and STModel classes simply define a bespoke data input and replace Model in ModelUI.

The basis of the SimpleTide model and how it can be applied is explained more fully in the SimpleTide manual (see the Docs folder in the SimpleTide source folder).

## Breach model

This model implements the methods outlined in Townend (2008a; 2008b) to estimate the regime section of a breach in a seawall that allows tidal flow into the area behind the seawall. The key components of this model are the site hypsometry (variation of plan area within the breached site as a function of elevation), the sediment erosion properties, the tide and wave conditions at the location of the breach. The model is implemented by defining five new classes: MRBreach, MRBreachModel, MRBreachData, MRSiteData and Hypsometry. The first of these overloads the ModelUI class, to provide a definition of the functionality required for this model whilst making use of the default functions where needed. The MRBreachModel class defines the specific model and replaces the Model class in ModelUI. The MRBreachData and MRSiteData classes handle the definition and display of the model variables and demonstrates the functions provided by the inherited PropertyInterface class. The Hypsometry class is additional and defines the properties and methods used to allow the user to interactively define the site hypsometry.

The basis of the MRBreach model and how it can be applied is explained more fully in the MRBreach manual (see the Docs folder in the MRBreach source folder).

## InshoreWaves model

This model uses an offshore wave timeseries to compute the inshore wave height at the edge of the surf zone. The model outputs include inshore wave height and direction and the depth of the edge of the surf zone. This model is included to demonstrate the handling of time series data using the timeseries Matlab functions and the DataSet interface class, which is included as part of ModelUI. It is implemented with four new classes: InWave, InWaveSite, InWaveData and InWaveModel. The class InWave inherits from ModelUI and provides the additional options needed. InWaveSite defines the nearshore site conditions. The additional classes handle the input wave timeseries (InWaveData), modelling of inshore waves (InWaveModel) and the modelling of deepwater waves (OffWaveModel).

The basis of the InWave model and how it can be applied is explained more fully in the InshoreWaves manual (see the Docs folder in the InshoreWaves source folder).

## Diffusion model

This model implements solves the 2-D diffusion equation based on the code developed by Suraj Shanka, Copyright (c) 2012 and made available via the Matlab TM Exchange Forum. The sole purpose here is to provide time varying 2-D data (and 3-D by replicating the 2-D matrix) to demonstrate the use of variables that have dimension of time and xyz. A Matlab Table can hold multiple variables for each time step, each with a consistent xyz definition (i.e. the size of the variable array at each time step is constant). The positions that define XYZ and Time are stored as properties of the table and each (multi-dimensional) variable is a column vector in the table. If the variable array remains constant but the positions (xyz) change with time, then xyz need to be added as a variable rather than as an xyz definition. The model is implemented by defining four new classes: Diffusion2D, DiffusionModel, DiffData and RunData. In addition, the function difffusion2Dmodel does the diffusion computations. DiffData and RunData classes are for data entry, illustrating how to separate out different aspects of the model input information. The Diffusion2D class inherits ModelUI to provide model specific functionality and the DiffusionModel class implements the control of model runs, tab plot display and holds the model results.

The basis of the Diffusion2D model and how it can be applied is explained more fully in the Diffusion2D manual (see the Docs folder in the Diffusion2D source folder).

## Derive Output

The *Run> Derive Output* option allows the user to make use of the data held within ModelUI to derive other outputs or, pass selected data to an external function (see Section 3.5). The equation box can accept t, x, y, z in upper or lower case. Time can be assigned to X, Y, or Z buttons or simply included in the equation as t (as long as the data being used in other variables includes a time dimension). Each data set is sampled for the defined data range. If the data set being sampled includes NaNs, the default is for these to be included (button to right of Var-limits is set to ‘+N’). To exclude NaNs press the button so that it displays ‘-N’. The selection is assigned to the variable limits whenever the current variable is assigned to X, Y or Z using the X, Y, Z buttons.

The equation string entered in the GUI is used to construct an anonymous function as follows:

heq = str2func(['@(t,x,y,z,utext,mobj) ',usereqn]); %handle to anonymous function

var = heq(t,x,y,z,utext,mobj);

This function is then evaluated with the defined variables for t, x, y, and z and optionally utext and mobj. utext allows text string to be passed (any string enclosed in single quotes, e.g. ‘Test’) and mobj passes the ModelUI handle.

Some functions may alter the length of the reference co-ordinates (x, y, z, t), or return more than one variable. These can be handled by passing a comment appended to the function definition e.g. subsample(x,t, thr,mobj) %time. In this version only ‘time’ is handled as a key word and this allows functions to derive a new timeseries with different time input values to those of the input variable(s).

If the function returns a single valued answer, this is displayed in a message box, otherwise it is saved, either by adding to an existing dataset, or creating a new one. There are three options:

1. Create a New Definition. The data type is ‘derived’, using the Derived class which is assigned to the h\_Derived class handle and the user defines the metadata specific to the instance (i.e. ResDef).
2. Create a New Case. This uses an existing Derived class instance and adds a new record to the selected instance using the existing Derived class instance variable definitions (i.e. ResDef).
3. Add to an Existing Case. This adds the variable to an existing Case by adding a new variable to the tscollection or table. The new variable definition is used to extend the existing Derived class instance variable definitions (i.e. ResDef).

Details of the implementation of these options is given in Section 6.8.6.

For vector data of the same length any range limits defined are applied to all the variables used and the result variable dimensions are also adjusted. Where one or more variables is an array, range limits are applied as follows:

1. Limits for the variable itself are applied by replacing values with NaNs;
2. Limits to the first dimension (time or x) are applied by removing rows that are outside the limits;
3. The dimensions of the array in each row are adjusted based on any limits set for the array dimensions.

Note: Limits are only set for dimensions that are included as variables in the equation.

Equations can use functions such as diff(x) - difference between adjacent values - but the result is n-1 in length and may need to be padded, if it is to be added to an existing derived data set. This can be done by adding a NaN at the beginning or the end: e.g.: [NaN;diff(x)]. NB: the separator needs to be a semi-colon to ensure the correct vector concatenation. Putting the NaN before the equation means that the difference over the first interval is assigned to a record at the end of the interval. If the NaN is put after the function, then the assignment would be to the records at the start of each interval.

*NB1: If the function returns multiple outputs only the first is used*.

*NB2: functions are forced to lower case (to be consistent with all Matlab functions), so any external user defined function call must be named in lower case.*

Another useful built in function allows arrays to be sub-sampled. This requires the array, z to be multiplied by an array of the same size. By including the dimensions in a unitary matrix, the range of each variable can be defined. For a 2D array that varies in time one way of doing this is:

>> z.\*repmat(1, length(t), length(x), length(y))

*NB3: the order of the dimensions t, x, y must match the dimensions of the array, z.*

This interface can also be used as an interface to user functions that are available in the Matlab search path. Simply type the function call with the appropriate variable assignment and the new variable is created. (NB: the UI adopts the Matlab convention that all functions are lower case). This is illustrated in the Diffusion2D model. The function userderivedoutput can be called with just a single variable defined as one of the results from the model to generate either the integral under the surface at each time step, or the surface gradients at each time step (see figure in Section 3.5 for inputs used).

For the integral option enter > userderivedoutput(t,x,y,z,’integral’)

For the gradient option enter > userderivedoutput(t,x,y,z,’gradient’)

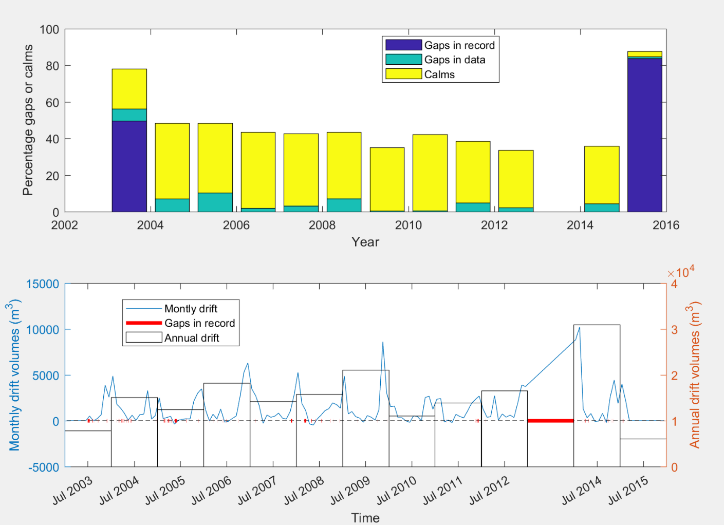
|  |  |
| --- | --- |
|  |  |

Some useful examples primarily for timeseries data include:

1. Moving Average. There are several moving average functions available from the Matlab Exchange Forum, such as moving.m. The call to this function is: < moving(X, n) > where n specifies the number of points to average over.
2. Down-sampling a time series. This allows a timeseries to be resampled at a different interval (that must be less than the source timeseries). The call to this function is:

<downsample(x, t, ’period’, ’method’)>, where x is the variable to be resampled, time is the associated time for that variable, period can be ‘year’, ’month’, ’day’, 'hour', 'minute', ‘second’, and method can be any valid function call such as ‘mean’, ‘std’, etc. The ‘period’ is required but the ‘method is optional and if omitted the mean is used.

For timeseries with gaps the ‘nanmean’ function is particularly useful but requires the Statistics toolbox.

1. Interpolate and add noise. To infill a record with additional points and, if required, add some random noise to the interpolated values. This is called using: <interpwithnoise(x, t, npad, scale, method, ispos)  %time>, where X is the variable, t is time, npad is the number of points to add between the existing data points, scale determines the magnitude of the random noise (a value of 0 results in an interpolated record with no noise), method is the Matlab algorithm used for the interpolation (the default is linear) and ispos is a true/false flag which sets negative values to zero if true.
2. Subsample one record based on a threshold defined for another record (e.g. subsample waves based on a threshold water level). Function is: <subsample(X, t, thr, mobj) %time>, where X and t are the variable to be subsampled, thr is the threshold value and mobj is the UI handle (must be mobj). The user is prompted to select the dataset and variable to be used to define the condition and a condition operator (<=, ==, etc). A time series is returned and added as a Derived data set. The user is prompted to define the metadata for the new data set.
3. Recursive plot. Generates a plot of a variable plotted against itself with an offset (e.g. x(i) versus x(i+1) ). This is called from the Derive Output GUI using: <recursive\_plot(x, ’varname’, nint)>, where x is the variable, ‘varname’ is a text string in single quotes and nint is an integer value that defines the size of the offset.
4. Phase plot. This function is similar to the recursive plot function but generates a plot based on two variables that can, optionally, be functions of time. The call to this function is: <phaseplot(X, Y , t)> where X and Y are the variables assigned to the respective buttons and t is time (this does not need to be assigned to a button and t can be omitted if a time stamp for the datapoints is not required).
5. Computing littoral drift statistics. Plots the annual and monthly volumes of drift along with details of gaps and calms. The call to this function is:

<littoraldriftstats(X, t, ’period’)>, where X is the rate of drift, time is the associated time for that variable and period can be ‘year’, or ’month’.

If no period is specified, the default is month. The period selection does not alter the plot (which shows both) but if the results are saved as a timeseries, period determines the timeseries interval. In the lower plot, the diamonds denote the start and end of the timeseries.

1. Compute the ratio of alongshore to cross-shore transport. The CERC formula for littoral transport is based on the energy flux (P) in the direction of wave advance per unit length of beach.

ie: F = P.cos(), where  is the angle between wave crest and bed contour. The longshore component of energy flux is P.cos().sin(), which leads to the main terms in the CERC formula. It follows that the cross-shore component is P.cos2(). The ratio of longshore/cross-shore energy flux (or transport potential) = tan(). The call to the function is: < beachtransportratio(X,theta)> where X is a timeseries of inshore wave directions and ‘theta’ is the angle of the shoreline to True North.

1. Generate a tidal range time series from a water level series using tidalrange(X, t) where X is the water level and t is the times of the water level values. Assumes that there are multiple water level values per tide. Also outputs mean water level and tidal range values as a table.
2. Plot a selection of frequency plots and histograms using waterlevelfreqplots(X, t) where X is the variable and t is time. Plot options include Water level elevation frequency, Water level spectrum, Elevations above a threshold, Duration of threshold exceedance, Elevation frequency above threshold. Designed to analyse water levels but could easily be adapted for other variables.

|  |  |
| --- | --- |
| Water level elevation frequency  1.5  a 0.5  -0.5  -1.5  Probability of occurrence (%) | 0.6  0.5  0.4  0.3  0.2  1.5  Duration frequency above 0.5 mOD  Duration (hours)  4.5  hr |

1. To add sea level rise (based on exponential growth from 1900 and zeroed to a defined year) to tidal water levels (ie predictions not measured water levels) the function:

<addslrtotides(X, t, delta, exprate, pivotyear)>, where X and t relate to the water level variable to be adjusted, delta is a rate for the year 1900 (e.g. 0.001 m/yr), exprate is the rate of exponential growth (e.g 0.011 for a fit to observations to-date) pivotyear is the year to use for zero sea level rise (e.g. 1900 adds slr based on change since 1900, whereas 2000 assumes that the tidal predictions are correct to the datum for the year 2000 and adjusts the record based on the slr function relative to that year).

The Function button on the Derive Output UI provides access to these predefined functions and allows the user to select one and load a function call template into the UI equation text box. The list is defined in the function functionlibrarylist.m which is in the MUIfunctions folder.

# Adding a new model to ModelUI

If the existing ModelUI interface is sufficient for your needs, then you can add models to work within the UI. This requires the model and the data input to be formatted in a particular way but then takes advantage of the generic interface provide by ModelUI. For details of how to implement a bespoke interface, rather than use the existing interface, see Section 6.

There are three steps needed to setup a new model within ModelUI:

1. Create a class for the data input;
2. Write the details of the model in a function file;
3. Generate the model specification within ModelUI.

These three steps are detailed in the following sections.

## Data input class

To define the input data required for a new model, create a new class with the desired properties. The file InputDataTemplate.m, in the Utilities folder, defines most of what is required. Copy the template to your working folder and rename the file with the name of the input data class (e.g. STData in SimpleTide). Then edit the file as indicated by the highlighted text in the template version of the file below. Replace all occurrences of InputDataTemplate with the name of your new class. Then add the properties needed by the model by renaming Variable1, Variable2, etc and similarly edit the variable descriptions as defined in the PropertyLabels property. This is all that is needed to use the popup form for data entry and to display the current data on the *Inputs* tab.

|  |
| --- |
| classdef InputDataTemplate < PropertyInterface  %class to handle model input data  %  %----------------------------------------------------------------------  % AUTHOR  % Ian Townend  %  % COPYRIGHT  % CoastalSEA, (c) 2017  %----------------------------------------------------------------------  %  properties (Hidden)  %abstract properties in PropertyInterface to define input variables  PropertyLabels = {'Variable 1 short description',...  'Variable 2 short description',...  'etc'}  %abstract properties in PropertyInterface for tab display  TabDisplay %structure defines how the property table is displayed  end    properties  Variable\_1 %definition of variable 1  Variable\_2 %definition of variable 2  end    %%  methods (Access=protected)  function obj = InputDataTemplate(mobj) %instantiate class  %constructor code: model specific properties for class  %uses the values defined in UI function setTabProperties  %inputs: obj-class handle; mobj-UI handle; <class name>  obj = setTabProps(obj, mobj,'InputDataTemplate');  end  end  %%  methods (Static)  function obj = setInputData(mobj,editflag)  %gui for user to set Input Data Property values  handle = getClassHandle(mobj,'InputDataTemplate');  if ~isa(mobj.(handle),'InputDataTemplate')  obj = InputDataTemplate(mobj);  else  obj = mobj.(handle);  end  %use PropertyInterface function to generate GUI  if nargin<2 || editflag  obj = editProperties(obj);  end  end  end  end |

Yellow highlights indicate edits or user input required to define new model

## Model function file

The Model class calls a user defined function which has three parts defined by a switch and three cases for ‘Model’, ‘Tab Plot’ and ‘Output Definition’. Copy the template to your working folder and rename the file with the name of the model function (lower case for a function is good practice, e.g. simpletidemodel in SimpleTide). Then edit the file as indicated by the highlighted text in the template version of the file below. Rename the function name ‘modelfunctiontemplate’ to be the same as the file name (the model function name). Add code for the Model as indicated by <code> to return the outputs defined at the end of this *case* block. The fields in the output structure for ‘results’ and one (or both) of ‘time’ and ‘xyzdata’ are required for the results to be accessible within ModelUI. Other fields allow additional information to be retained. For more details of data format definition required see

|  |
| --- |
| function output = modelfunctiontemplate(mobj,flag,plotdata)  %test loading complete model definition from a single file  switch flag  case 'Model'  %run model and return results to Model.runModel    %MODEL code or call to model function  %should define results,time,xyzdata,addnprops,metatxt  %if not required define as empty eg.'time',[],...  <code>  %output required is: results,time,xyzdata,addnprops,metatxt  %xyzdata values are returned as cell arrays for {x,y,z}  output = struct(...  'results',{results},... %cell array of variables  'time',(Bartlett et al.),... %time data (eg in seconds)  'xyzdata',{xyzdata},... %x,y,z data as required  'addnprops',{addnprops},...%additional output properties  'metatxt',{metatxt}); %variable specific text  %%  case 'Tab Plot'  %model results and additional properties passed in for plotting  %on the Plot tab  output = []; %dummy output - not required for plot  %  %code to generate plot on Plot tab  %variables are passed in using plotdata structure:  %plotdata.xy - XY or XYZT data to be plotted  %plotdata.addnprops - any additional properties saved  %plotdata.src - handle to plot tab  %plotdata.casehandle - handle to plot case (not record)  %plotdata.casedesc – metatext for selected data set  <code>    case 'Output Definition'  %output definitions used in Model class constructor  %see ConstantData Class for more detailed definitions  r = mobj.Constants.ResDef;  r.DataVar = {'ModelData'}; %property for model data in Model class  r.AdnOutVar = {'AddnProps'}; %property for additional properties  r.varNames = {'var1','var2','etc'}; %variable names  r.varDesc = {'var1desc','var2desc','etc'}; %variable descriptions  r.varUnits = {'units1','units2','etc'}; %variable units  r.varLabels = {'var1label','var2label','etc'};%variable plot axis labels  r.xyzDesc = {'Xdesc','Ydesc','Zdesc'}; %XYZ descriptions  r.xyzUnits = {'Xunits','Yunits','Zunits'}; %XYZ units  r.xyzLabels = {'Xlabel','Ylabel','Zlabel'}; %XYZ axis plot label  r.rowDesc = {'Time'}; %key word used to identify time data  r.rowUnits = {'time units'}; %time units (eg 'd','h','s')  r.rowLabels = {'Time (s)'}; %time axis plot label  r.rowFormat = {'dd-MM-uuuu HH:mm:ss.SSS'}; %format used for time data  r.xyDefault = {'Time'}; %default X-axis; can be'Time','X','Y',or Z'  r.Type = {'model'}; %data type  r.Style = {'Single'}; %write results to Excel spreadsheet  output = r;  end  end |

Yellow highlights indicate edits or user input required to define new model

The code for the plot to be produced on the *Plot* tab is defined in the ‘Tab Plot’ case. The data to be used in the plot is passed to the function in ‘plotdata’, a structure with fields containing the model data, any additional properties, the handle to the *Plot* tab and the handle to the case being used. See simpletidemodel.m and verticalprofile.m for examples of tab plots.

The final *case* block is for the ‘Output Definition’. This defines the ResDef structure property in the Model class. The fields within the structure define the properties of the variables, xyz data and time to be used when handling the model output. The template for the ResDef structure is defined in the ConstantData class and details of the fields within ResDef are detailed in Section 6.8.2.

### Model output formats

The correct format to create a dscollection (table) or tscollection depends on the number of dimensions to be handled and how the data is to be stored. The following examples outline the most common usages:

1. Variable and X

results – {vector for each variable to be saved}

time – {}

xyzdata – {vector for X} passed as a cell array

addnprops – {cell arrays of any additional properties to be stored (e.g. run statistics)}

metatxt – {cell array of metadata text stored with each variable array in ‘results’}

1. Variable and X, Y (and Z)

results – {matrix with rows=1 and additional dimensions to match X,Y,Z. Use [m,n] = size(z);

reshape(z,1,m,n); to format}

time – {}

xyzdata – {vectors for X, Y (and Z)} passed as a cell array

addnprops – {cell arrays of any additional properties to be stored (e.g. run statistics)}

metatxt – {cell array of metadata text stored with each variable array in ‘results’}

1. Variable and Time

results – vector or matrix with row length matching the time vector

time – vector for time in format defined in <Model Class>.ResDef property

[*NB: call to DataSet.setCollection uses ‘Time’ for tscollections and ‘rowNames’ for dscollections (tables)]*

xyzdata – {}

addnprops – {cell arrays of any additional properties to be stored (e.g. run statistics)}

metatxt – {cell array of metadata text stored with each variable array in ‘results’}

1. Variable and Time and X, Y (and Z)

results – {matrix with rows for time and additional dimensions to match X,Y,Z}

time – vector for time in format defined in <Model Class>.ResDef property

xyzdata – {vectors for X, Y (and Z)} passed as a cell array

addnprops – {cell arrays of any additional properties to be stored (e.g. run statistics)}

metatxt – {cell array of metadata text stored with each variable array in ‘results’}

## Model specification in ModelUI

With ModelUI open (in any model) use *Setup>Model Specs* to edit, add or delete a model specification. The first list dialogue allows the user to either select an existing model specification to edit, to add a new one, or to delete one. When the edit option is selected the pop-up, dialogue presents the existing definition for editing. When adding a new model specification, a similar blank dialogue is used to provide the details of the calls to be used. If delete is selected, then a second list dialogue prompts the user to select the existing model specification for deletion.

The dialogue requires details of the model implementation including name, description, log and help files. In addition, the call used for various actions have to be defined. The example below shows the Vertical Profile model specification. Data is input using the VPInputData class. For a new model a new class for data input will be defined and so this will need to be specified here. Similarly, if you add a bespoke class for adding Imported Data (e.g. Observations) then this will also need to be specified, otherwise the default can be used. If the Model class is used, only the name of the model function needs to be changed and the other calls remain the same. However, if a bespoke model class is used then the various calls will need to be specified.

The model specification file is read from and saved to the top level ModelUI folder with the name: ModelUImodelSpecification.mat. NB: if the file is not found, the user is prompted to select a folder where the file can be found. For this file to be found when running ModelUI, the head folder needs to be on the Matlab path. This will be the case if ‘addModelUIpaths’ is run at the beginning of the session, or during Matlab start-up (see Section 2.1).

|  |  |
| --- | --- |
|  | Name of model  Description of model  Format is ClassName.Call to the function to input the model data  Format is ClassName.Call to the function to input reference data (e.g. input time series data or observed data sets).  Format is ClassName.Call to the function to run the model  Name of model function  Data to be saved as a table or a timeseries (tscollection)  Name of the model logo file (if used)  Name of the model help file |

## Models provided to run within ModelUI

1. Vertical Profile

The Vertical Profile model is implemented within ModelUI and uses the default UI. The VerticalProfile sub-folder contains the files needed to set up the model:

VPInputData – the class file for model data input

verticalprofilemodel – the model function file

VerticalProfileData – class to load vertical profile data (z, u) from a file (e.g. to compare with model)

Logo and Help files are also provided, although the default ModelUI files could be used instead. The runverticalprofile file runs the model within ModelUI.

1. Simple Tide

The Simple Tide model is also implemented within ModelUI and uses the files:

STData – the class file for model data input

simpletidemodel – the model function file

Again, there are model specific Logo and Help files. This is all that is required to run the model within ModelUI.

## Alternative method to set up a Model using the default interface

Rather than adding a Model Specification within ModelUI, as described in Section 5.3, a new UI class that inherits from ModelUI can be used to add a new model. This is illustrated using SimpleTide which includes the following files to implement the UI and model call:

SimpleTide – class that inherits from ModelUI to implement the model specific functionality

STModel – class to implement the model. This replaces the Model class that is used in the generic ModelUI and in this case includes the model specific functions.

runsimpletide – a script to run the bespoke version of SimpleTide (i.e. not the version that uses simpletidemodel from within ModelUI).

The code for the UI is as follows:

|  |
| --- |
| classdef SimpleTide < ModelUI  % main GUI for SimlpleTide which inherits from ModelUI and implements  % GUIinterface to define the main menus.  %  properties  vNumber = '2.0'  vDate = 'Jan 2019'  %additional handles to those defined in ModelUI  %STData class uses h\_InputData  %STModel class uses h\_Model  %WaterLevelData class uses h\_ImportedData  end  %%  methods (Static)  function obj = SimpleTide  %constructor function initialises GUI  %NB setGUI is called by ModelUI constructor  end  end  %%  %----Functions called by menu->  % The following functions overload functions in ModelUI to redirect  % calls to the SimpleTide input and model classes (STData and STModel)  % need to maintain the same access permissions used in ModelUI  methods  function obj = setGUI(obj,~)  %initialise standard figure and menus  obj.FigTitle = sprintf(sprintf('%s - Version: %s; Date: %s',...  'Simple Tide',obj.vNumber,obj.vDate);  obj.ModelLogo = 'SimpleTide\_logo.png';  obj.ModelHelp = "SimpleTideHelp.txt";  obj.ModelInputs.STModel = {'h\_InputData'};  obj.SaveDataString = 'h\_InputData';  obj.NumTabs.Plot = 2; %1=XY only; 2=XY,XYZ; 3=Animate  obj.NumTabs.Stats = 2; %General+Timeseries+Taylor  obj = initialiseUI(obj); %initialise menus and tabs  %ModelSpec for ModelInput, DataInput and RunModel as required  obj.ModelSpec.ModelInput = 'STData.setInputData';  obj.ModelSpec.DataInput = ' WaterLevelData.loadData';  obj.ModelSpec.RunModel = 'STModel.runModel';  %if required call setAdditionalMenus(obj) here  end  %%  function obj = setClassHandles(obj)  %cell array the defines the assignment of each class used in  %the model to a handle. Format {handle name, class name;...}  modelhandles = {'h\_InputData','STData';...  'h\_ImportedData','WaterLevelData';...  'h\_Model','STModel'};  obj.ModelHandles = vertcat(obj.ModelHandles,modelhandles);  %setClassHandles is called by GUIinterface.setModelHandles  %where obj.ModelHandles is converted to a table with variables  %handle and class  end  %%  function props = getTabProperties(~)  %define the tab and position to display class data tables  %props format: {class name, tab tag name, position, column width}  props = {...  'STData','Inputs',[0.95,0.48],{180,60},'Input data:';... %InputData  'InputFile','Inputs',[0.25,0.95],{80,420},'Input files:'}; %InputFile  end  end  end |

The yellow highlights identify the code that is model/UI specific. The green highlights are the class.function calls for the input of data and running of the model.

For this approach to customising the UI, the three components of the model function file (Section 5.2) are written as functions in a new ‘model’ class, as illustrated by the STModel class.

Whilst both versions of SimpleTide use the standard ModelUI interface, the other three models (InshoreWaves, MRBreach and Diffusion2D) all customise the ModelUI interface in some way and so are only provided as stand-alone models, which are created as explained in Section 6.

# Implementing a new bespoke model

Examine the program structure to understand how information is passed between classes and methods (see Section 7 for details of Program structure). Then define the behaviour you want, which should include the following workflows:

* Model setup: data input, and any runtime settings or conditions, and how this setup information should be displayed to the user on one or more tabs;
* Any additional information or setup the functionality required to help the user (which may require additional tabs). See, for example, the Hypsometry class in the MRBreach model, which allows the user to interactively alter and select the hypsometry to be used in the model run;
* The behaviour required at runtime;
* What results should be displayed using a popup dialogue box or a tab and what results should be saved, so that they can be plotted, or exported.

The demonstration models included as part of the ModelUI package, illustrate alternative types of model implementation.

* All the stand-alone models overload ModelUI with a new UI class that inherits ModelUI. This is to redirect the UI menu options to the model specific classes and provide any additional functionality required. As explained in Section 6.1, this is not a requirement but does ensure that the models are more robust.
* The SimpleTide model simply replaces the Model class in ModelUI with a bespoke model class, as explained in Section 6.1.
* The MRBreach model splits the input data into two classes (MRBreachData and MRSiteData) and adds an additional tab for Hypsometry, as explained in Section 6.2. The Diffusion2D model does something similar using classes DiffusionData and RunData and implements menu options for Input Data and Run Parameters.
* The handling of time series data (using the Matlab timeseries functions) requires the use of the DataSet class and this is described in Section 6.4 and is illustrated in the InshoreWaves model.
* Finally, the handling of multi-dimensional variables (e.g. time and x-y-z) is implemented using Matlab tables as explained in Section 6.6 and is illustrated in the Diffusion2D model.

## Using ModelUI for a different model

The simplest way to adapt ModelUI for a different application is to overload the ‘InputData’ and ‘Model’ classes. This was how SimpleTide was originally implemented and worked fine providing it was run from the SimpleTide folder and that no other ModelUI models were already open. To avoid conflicts and be able to run several models at the same time, it is better to implement classes with unique names so that the code can find the correct class. This however requires, as a minimum, the redirection of menu calls in the main user interface[[2]](#footnote-2). The following section explains how to do this, and the subsequent section looks at defining new classes for the input data and model. To begin, create a new folder for the model. For the implementation of SimpleTide this is located in ../SimpleTide relative to the ModelUI folder.

### The User Interface

For the InWave model, the InWave class inherits from ModelUI. If there were no additional methods, it would simply implement the same functionality as ModelUI. The functions that are included in InWave.m, modify the behaviour of functions in ModelUI.m.

The file starts with the class definition:

classdef InWave < ModelUI

and has an accompanying end statement at the end of the file. The file name is the same as the class name and ‘< ModelUI’ indicates that InWave inherits from the ModelUI class. This is followed by the definition of any additional properties and the constructor function for InWave.

properties

vNumber = '2.0'

vDate = 'Jan 2019'

%additional handles to those defined in ModelUI (format h\_\*\*\* required)

%InWaveSite class uses h\_InputData

%InWaveModel class uses h\_Model

h\_InWaveData %handle to class to handle measured wave data

h\_WaterLevelData %handle to class for water level data

h\_OffWaveModel %handle to classs that defines deepwater wave model

h\_WindData %handle to class for measured wind data

h\_WaveHindcast %handle to class for wind-wave model output

h\_WindWaveInput %handle to class for wind-wave model input parameter

end

%%

methods (Static)

function obj = InWave

%constructor function initialises GUI

%NB setGUI is called by ModelUI constructor

end

end

When InWave is called this first initialises the parent, i.e. ModelUI. Within the ModelUI constructor is a call to setGUI which defines a number of model specific properties. There is therefore no need to call this again from the InWave constructor. However, there is a need to overload the function setGUI with a version specific to InWave. This is as follows:

function obj = setGUI(obj,~)

%initialise standard figure and menus

obj.FigTitle = sprintf('%s - Version: %s; Date: %s',...

'Inshore Waves’,obj.vNumber,obj.vDate);

obj.ModelLogo = 'InshoreWaves\_logo.png';

obj.ModelHelp = 'InWaveHelp.txt';

obj.ModelInputs.InWaveModel = {'h\_InputData','h\_InWaveData'};

obj.ModelInputs.OffWaveModel = {'h\_InputData','h\_InWaveData'};

obj.ModelInputs.WindWaveModel = {'h\_WindWaveInput','h\_WindData'};

obj.SaveDataString = 'h\_InputData';

obj.NumTabs.Plot = 2; %1=XY only; 2=XY,XYZ; 3=Animate

obj.NumTabs.Stats = 3; %General+Timeseries+Taylor

obj = initialiseUI(obj); %initialise menus and tabs

%if required call setAdditionalMenus(obj) here

setAdditionalMenus(obj);

end

The yellow highlights indicate the changes from the function used in ModelUI.

This now loads the model specific version number and date, logo file and help file (NB these need to exist in the model folder). Some of the other models illustrate how to use different class handles.

To map the classes used in the model to the class handles for the bespoke UI the function setClassHandles defines the mapping. The function for InWave is as follows:

|  |
| --- |
| function obj = setClassHandles(obj)  %cell array the defines the assignment of each class used in  %the model to a handle. Format {handle name, class name;...}  modelhandles = {'h\_InputData','InWaveSite';...  'h\_ImportedData','';...  'h\_Model','InWaveModel';...  'h\_InWaveData','InWaveData';...  'h\_WLData','WaterLevelData';...  'h\_OffWaveModel','OffWaveModel';...  'h\_WindData','WindData';...  'h\_WaveHindcast','WindWaveModel';...  'h\_WindWaveInput','WindWaveInput'};  obj.ModelHandles = vertcat(obj.ModelHandles,modelhandles);  %setClassHandles is called by GUIinterface.setModelHandles  %where obj.ModelHandles is converted to a table with variables  %handle and class  end |

The yellow highlights are the default handles defined for ModelUI. For InWave, two of these area used for the InWaveSite and InWaveModel classes. The third is included in the cell array but left undefined as it is unused in this model. The green highlights are three additional classes used by InWave and their associated handles. In general, additional handles use the class name with the prefix ‘h\_’.

The definition of what data sets are displayed and on which tab is defined in the getTabProperties function. If the InputData or InputFile classes have been replaced with other classes or additional classes have been added this function needs to be overloaded with a UI specific version, e.g.:

|  |
| --- |
| function props = getTabProperties(~)  %define the tab and position to display class data tables  %props format: {class name, tab tag name, position, ...  % column width, table title}  props = {...  'InWaveSite','Inputs',[0.95,0.62],{250,60},'Site data:';...  'InputFile','Inputs',[0.44,0.95],{80,420},'Input files:';...  'WindWaveInput','Inputs',[0.95,0.97],{160,50},'Wind-wave parameters:'};  end |

In addition, there is a need to overload some of the menu functions and their callback functions (e.g. inputProps and runModel).

A modified vesion of the setupMenu is included to remove the ‘Model Specs’ option that is included in the default ModelUI version.

The setAdditionalMenus modifies the menu drop down to include options for Wave data and Water level data. The dataSubMenu provides a sub-selection for the Load, Add and Quality control options. The inputdataSubMenu then calls the callback function for Site Data and Model Constants but passes to the sub-menu for Wave and Water level data (Load, Add, etc).

The inputProps function defines the response to each menu selection in the Setup menu. The Switch cases correspond to the menu options (NB the character strings must match). For each case the classCallFunction is called and this requires the name of the class being called, the name of the function the class that is being called and a flag. The flag is true if the input data are to be displayed on a tab, in other cases the flag is false (including when data sets are being loaded from a file and the file name is displayed on a tab using the PropertyLink property.

The changes for running a model are similar. The runMenu function is overloaded to include option to the run the Inshore and Deepwater models (as well as the Derive Output option). All options use the runModel function and this also uses the classCallFunction to direct the interface to the required class and class function. In this usage the flag is not required. In summary, the main changes required are to add (or remove) menu items and then define the class and function that each menu item should call.

Thus, the changes required to establish a new UI with different input and model definitions are relatively minor and most of the code can be cut and pasted. The file ‘NewUItemplate.m’, in the Utilities folder, provides a starting template for creating a new UI, which includes several of the ModelUI functions that typically need to be overloaded to create the desired UI.

### Data Input and Model classes

1. Defining a class for data input

InWave uses the InWaveSite class for data input. This is the same as explained in Section 5.1 and the file InputDataTemplate.m, in the Utilities folder, defines most of what is required.

An input data file contains the following components:

* A class definition that inherits the PropertyInterface class

classdef InWaveSite < PropertyInterface

* The definition of properties that control the data input and display.

properties (Hidden)

%abstract properties in PropertyInterface to define input variables

PropertyLabels = {'Offshore bed level (mOD - so probably negative) ',...

'Friction Coefficient (value 0-1)',...

'Shoreline Angle (degTN)',...

'Beach crest or HW level (mOD)',...

'Upper beach slope (1:s - enter value for s)',...

'Bed level 1km out from SWL (mOD)',...

'Inshore bed level NB: NaN if not used',...

'Wave breaking model (0,1,2, or 3 - see manual)'}

%abstract properties in PropertyInterface for tab display (not used)

TabDisplay %structure defines how the property table is displayed

end

The variables in this property block have the following uses:

PropertyLabels: lists the variables requiring data input. These labels are used in the UI for data entry and should match the list of variables defined in the model variables property block (see below).

TabDisplay is a structure with three fields:

PropertyTab: defines which tab to use to display a table of the currently active input data

TabPosition: defines the position of the input data table on the ‘PropertyTab’ tab (uses normalized units relative to tab). Some typical TabPosition definitions are:

top left → [0.95,0.48]; top right → [0.95,0.95];

lower left →[0.45,0.48]; lower right → [0.45,0.95];

TabColWidth: columns widths used in the display of model variables on the ‘PropertyTab’ tab (uses pixel units).

The tab values are defined in the model UI setup (e.g. ModelUI) in the function getTabProperties and initialised in the input data class constructor (see Section 6.3). This allows input data to be displayed on different tabs to suit the model UI.

The definition of the properties to be used in the model (this depends on the model requirements)

properties

OffshoreBedLevel %offshore bed level (OD-water depth) (mOD)

FrictionCoefficient %friction coefficient (default=1)

ShorelineAngle %angle of contours fron north (degrees TN)

BeachCrestLevel %beach crest level (mOD)

UpperBeachSlope %bed slope (1:bs)

BedLevelat1km %bed level 1km out from SWL (mOD)

InshoreBedLevel = NaN %inshore bed level (mOD) - overrides use of surf depth

WaveBreakingModel = 1; %flag to indicate which wave breaking model to use

%0=no breaking, 1=SPM breaking on a slope,

%2=Le Roux, 3=Hsb (Tucker/Holmes)

end

* Add the functions InWaveSite, to instantiate the class, and setInputData. The latter is the same as the equivalent function in the InputData class but with the following modifications highlighted in yellow:

methods (Access=protected)

function obj = InWaveSite(mobj) %instantiate class

%constructor code: model specific properties for STData class

%uses the values defined in UI function setTabProperties

obj = setTabProps(obj,mobj,'InWaveSite');

end

end

%%

function obj = setInputData(mobj,editflag)

%gui for user to set Input Data Property values

h\_name = getClassHandle(mobj,'InWaveSite');

if ~isa(mobj.(handle),'InWaveSite')

obj = InWaveSite;

else

obj = mobj.(handle);

end

%use PropertyInterface function to generate GUI

if nargin<2 || editflag

obj = editProperties(obj);

end

end

(NB: the version of InWaveSite included is slightly different to the sample code above because it uses a singleton class).

1. Replacing the Model class

Create a new **InWaveModel** class to replace the **Model** class in the ModelUI model, with the following components:

* A class definition that inherits the DSDataSet class if you want to use tables

classdef Model < DSDataSet

OR

* A class definition that inherits the TSDataSet class if you want to use timeseries

classdef InWaveModel < TSDataSet

The definition of properties that control the handling of the model results is almost the same in DSDataSet and TSDataSet, comprising ResDef, InputHandles and a property to hold the results (the default is ModelData in DSDataSet and mtsc in TSDataSet). ResDef holds the information to define the results. The properties block also includes the InputHandles property. This is a cell that defines all the classes to be checked in the call to isValidModel (N.B. this only works for classes that use the PropertyInterface), e.g.: InputHandles = {'h\_InputData'}. In InWaveModel there are two additional properties:

InputDataCISs holds the Case IDs for the wave and water level data that are used for the model run. InshoreBedSlope holds the bed slope at the inshore point(s).

Create a ‘methods’ block and add a constructor function with a name that matches the model class name. Within the constructor function, define the variables in ResDef required for the particular model. The full list is given in Section 6.8.2 and the code below shows those used in InWaveModel. These are required to handle the correct storage and retrieval of model data and to pass descriptions of the data to utilities such as plotting. In this case the property that holds the data set is defined along with a definition of the variables in the data set and the independent variable, time. The time variable is held as row definitions in the table. The remaining variables define:

xyDefault – the variable for the X-axis in XY plots;

Type – the type of data being saved (e.g. ‘model’ for model results and ‘data’ for imported data sets);

Style – the format for saving to an Excel spreadsheet (Single or Multiple).

The constructor function instantiates the model and initialises ResDef:

methods (Access=private)

function obj = InWaveModel(mobj)

%constructor to initialise object

r = mobj.Constants.ResDef;

r.DataVar = {'mtsc'};

r.AdnOutVar = {'InshoreBedSlope'};

r.varNames = {'Hsi','Diri','swl','depi'};

r.varDesc = {'Inshore wave height','Inshore wave direction',...

'Still water level','Inshore depth'};

r.varUnits = {'m','deg','mOD','m'};

for i=1:length(r.varDesc)

r.varLabels{i} = sprintf('%s (%s)',r.varNames{i},r.varUnits{i});

end

r.rowDesc = {'Time'}; %used to identify time data

r.rowLabels = {'Time'};

r.rowFormat = {'dd-MM-uuuu HH:mm:ss'}; %using MM avoids problems with ‘Locale’

r.xyDefault = {'Time'}; %can be 'Time','X','Y',or Z'

r.Type = {'model'}; %'model' or 'data' currently handled

r.Style = {'Single'};

obj.ResDef = r;

%Set InputHandles for input data required by model

obj.InputHandles = mobj.ModelInputs.InWaveModel;

end

end

* Create a Static methods block and add the function runModel, which should contain the code to run the model (or call other functions that run the model).

|  |
| --- |
| methods (Static)  %--------------------------------------------------------------------------  % Model implementation  %-------------------------------------------------------------------------  function obj = runInWaveModel(mobj)  %check to see if there are existing models and add record  h\_model = getClassHandle(mobj,'InWaveModel');  if isa(mobj.(h\_model),'InWaveModel')  obj = mobj.(h\_model);  irec = length(mobj.(h\_model).mtsc)+1;  else  obj = InWaveModel(mobj); %instantiate object  irec = 1;  end  %now check that the input data has been entered  %isValidModel checks the InputHandles defined in InWave  if ~isValidModel(mobj,obj.InputHandles)  warndlg('Use Setup to define model input parameters');  return;  end  %--------------------------------------------------------------------------  % Model code  %--------------------------------------------------------------------------  %individual properties are retrieved as obj.<property>  [code block for model]  %assign the model results to the timeseries collection, tsc  %using the Time property triggers the use of a tscollection  tsc = DataSet.setCollection(obj,[Hsi,Diri,swl,depi],...  'Time',datetime(Time),'metaData',{meta.inputxt});  if isempty(tsc), return; end %trap error before assigning to obj  %get the Case IDs of the input data  h\_wave = getClassHandle(mobj,'InWaveData');  InWaveCID = getCaseID(mobj.Cases,h\_wave,meta.ids);  h\_wl = getClassHandle(mobj,'WLData');  InWLevelCID = getCaseID(mobj.Cases,h\_wl,meta.idw);  %--------------------------------------------------------------------------  % End of Model code  %--------------------------------------------------------------------------  %The order of the following calls is important. Results needs  %mobj to have been updated with the new data set  %assign data set to class data property  obj.mtsc{irec} = tsc; %assign to master tscollection for class  %store source CIDs - enables other models to retrieve input data  obj.InputDataCIDs{irec} = [InWaveCID,InWLevelCID];  obj.InshoreBedSlope{irec} = bs; %can be an array or single value  %save pointers to results for the data set loaded  recnum = Results.saveResults(mobj,h\_model,obj);  %add description to the tsc.Name variable  obj.mtsc{irec}.Name = mobj.Cases.CaseDescription{recnum};  ModelUI.myDialog('Run complete');  DrawMap(mobj);  end |

Highlighted text again indicates the changes made that are specific to this model.

In this model the data are timeseries and the model inherits the DataSet class. This uses tscollections assigned to the property obj.mtsc.

If a table were to be used (as in SimpleTide) a property such as ModelData is used to hold the data. Typically, ‘obj.ModelData’ is an array where each output is a column vector (of the same length) and these are concatenated as an array.

Adjust the assignment of obj.mtsc or obj.ModelData to capture the model output vectors that you want to save.

In this model the results are saved as a Matlab timeseries collection and this used the ‘Time’ keyword in the call to DataSet.setCollection. If the keyword ‘rowNames’ is used to pass the time variable, data are stored in a Matlab table.

The last block stores details of the model run as a model scenario using the Results class.

* Add the function tabPlot, to include a plot on the Plot tab. The InWaveModel does not have a tabPlot function because it uses the default timeseries tabTSPlot plotting function in TSDataSet class. There is also a default plotting function, tabDSPlot in the DSDataSet class.

|  |
| --- |
| function tabPlot(obj,src,~,inp)  %generate plot for display on Plot tab from Table data  %data is retrieved by GUIinterface.getTabData  propid = 1; %id for property that holds the data in obj.ResDef.DataVar cell  selecteddata = obj.(inp.dprop{propid}){inp.id\_rec};  %-------------------------------------------------------------  <*extract the variables required e.g:>*  p.t = getRowNames(obj,selecteddata);  if isempty(p.t)  default = obj.ResDef.xyDefault;  idx = strcmp({'X','Y','Z'},default);  p.t = selecteddata.Properties.UserData.XYZ{idx};  end  p.z = selecteddata{:,1};  [code block for tab plot]  end |
|  |

This function checks to see if any results have been written to the Cases handle. If not, the code checks that some input data exists and then runs the model.

If a plotTab is not required, it is advisable to include the following code to avoid error warnings if the tab is clicked on:

|  |
| --- |
| methods  function tabPlot(obj,src,~,inp)  %generate plot for display Plot tab  return;  end  end |
|  |

Alternatively, overload the setAppTabs function that can be found in ModelUI.m. To do this, add the function setAppTabs to the model UI initialisation code (in this case InWave.m) and delete the code that initialises the unwanted tab, i.e.:

|  |
| --- |
| %initialise tabs for specifc model  uitab(gobj.GuiTabs,...  'Title',' Plot ','Tag','Plot',...  'ButtonDownFcn',@gobj.getTabData); |
|  |

## Data input

In the MRBreach class, the menu option for Model Data calls setInputData in the class MRBreachData. This class only handles the input and editing of the model data and so illustrates the main functions of the PropertyInterface which MRBreachData inherits from. MRBreachData comprises two property blocks, one of which is hidden, and two functions in the methods block. The hidden property block defines properties that control the data input and display, as explained above in Section 6.1.2. The second property block defines the variables to be used. The controlling UI calls setInputData which instantiates the class using getInputData function. This is implemented differently to some of the other input data classes (e.g. STData) because a singleton class is used. This means that there can only be one version of the model input properties at any one time. The setInputData function then calls editProperties which is a function inherited from the PropertyInterface.

The MRBreach model also illustrates the use of multiple input data classes, with the addition of the MRSiteData class. The format of the class is the same as MRBreachData and just the properties and their respective definitions are changed. In this case the PropertyTab is kept the same as for MRBreachData as ‘Inputs’ but the TabPosition is adjusted so that the SiteData table is alongside the MRBreachData table on the *Inputs* tab. By amending the PropertyTab and creating an additional tab in MRBreach this could have also been separated into two tabs. Which option to adopt usually depends on how may input variables there are in each class.

The Diffusion2D model also has two classes to handle the model inputs, one to define the model domain (DiffusionData) and one to handle the run time parameters (RunData). These are accessed using a slightly different menu layout under Setup, which is defined in the Diffusion2D class.

## UI customisation

A simple adaptation of the model UI was outlined in Section 6.1.1. Here we look at a more extensive modification to use different class handles and add an additional tab. This uses MRBreach as an example.

Create a model specific user interface by overloading the ModelUI class. (If the extensions are extensive, it may be simpler to inherit from the interface, ‘GUIinterface’, directly). You then need to define the properties you want for your model and add the method that instantiates the object (the class constructor). In the example models the constructor simply inherits this structure and redefines the class name from ModelUI to AnotherModel. The specific model interface is defined using the call to setGUI, as for example in MRBreach:

|  |
| --- |
| function obj = setGUI(obj,~)  %initialise standard figure and menus  obj.FigTitle = sprintf(sprintf('%s - Version: %s; Date: %s',...  ‘MRBreach’,obj.vNumber,obj.vDate);  obj.ModelLogo = 'MRBreach\_logo.png';  obj.ModelHelp = 'MRBreachHelp.txt';  obj.ModelInputs.MRBreachModel = {'h\_InputData','h\_MRSiteData'};  obj.SaveDataString = 'h\_InputData';  obj.NumTabs.Plot = 2; %1=XY only; 2=XY,XYZ; 3=Animate  obj.NumTabs.Stats = 1; %General stats  obj = initialiseUI(obj); %initialise menus and tabs  %if required call setAdditionalMenus(obj) here  end |

The first 3 highlighted lines simply change the UI name, and point to model specific splash image and text help file. The 4th highlighted line then defines the input classes that are required to be able to run the model (defined using their class handles). SaveDataString is the handle that is checked to see whether there is a project to be saved when exiting the UI. The NumTabs property defines how many tabs to be included from the Plot and Statistics UI. The call to initialiseUI then initialises all the menus and tabs.

Other options can be defined in setGUI

To map the classes used in the model to the class handles for the bespoke UI, the function setClassHandles defines the mapping. The function for MRBreach is as follows:

|  |
| --- |
| function obj = setClassHandles(obj)  %cell array the defines the assignment of each class used in  %the model to a handle. Format {handle name, class name;...}  modelhandles = {'h\_InputData','MRBreachData';...  'h\_ImportedData','';...  'h\_Model','';...  'h\_MRSiteData','MRSiteData';...  'h\_MRBreachModel','MRBreachModel';...  'h\_Hypsometry','Hypsometry'};  obj.ModelHandles = vertcat(obj.ModelHandles,modelhandles);  %setClassHandles is called by GUIinterface.setModelHandles  %where obj.ModelHandles is converted to a table with variables  %handle and class  end |

One of the calls from the class constructor is to the setAppTabs function. If you want to add tabs or change the behaviour of the default tabs you will need to write a new version of this function in your new UI class. The following is an example of a modified tab behaviour taken from MRBreach, where the Hypsometry tab has been added.

|  |
| --- |
| function setAppTabs(gobj)  %default tabs for Scenarios (list of model runs), summary of  %current model parameters and Help  uitab(gobj.GuiTabs,...  'Title',' Inputs ','Tag','Input',...  'ButtonDownFcn', ...  @(src,evdat)Project.ModelSummary(gobj,src,evdat));  %initialise tabs for specific model - Hypsometry  uitab(gobj.GuiTabs,...  'Title',' Hypsometry ','Tag','Hypsometry',...  'ButtonDownFcn', ...  @(src,evdat)Hypsometry.tabHypsometry(gobj,src,evdat));  %Plotting  uitab(gobj.GuiTabs,...  'Title',' Plot ','Tag','Plot',...  'ButtonDownFcn',@gobj.getTabData);  %default tab for Help  uitab(gobj.GuiTabs,...  'Title',' Help ','Tag','Help',...  'ButtonDownFcn',...  @ModelUI.Help);  end |

The Stats UI produces plots and tables. The tables can be plotted as stand-alone figure, or assigned to a tab. To assign results to a tab the setAppTabs function needs to have a Stats tab added. The following code adds the Stats tab and two sub-tabs for the Descriptive and Extreme statistical table outputs. The ButtonDownFcn calls both call a function in the DataStats class that gets the statistical results and generates the output table.

|  |
| --- |
| %statistics table results tab  statstab = uitab(gobj.GuiTabs,'Title',' Stats ','Tag','Stats');  %sub tab options for Stats tab  stgstats = uitabgroup(statstab,'Tag','SubGuiTabs');  uitab(stgstats,'Title',' Descriptive ','Tag','Descriptive',...  'ButtonDownFcn',@(src,evdat)DataStats.statsTab(gobj,src,evdat));  uitab(stgstats,'Title','Extremes ','Tag','Extremes',...  'ButtonDownFcn',@(src,evdat)DataStats.statsTab(gobj,src,evdat)); |

If you want to change where input data is displayed, or if you have changed the names of the tabs, you will need to modify the function setTabProperties. The ‘props’ variable is a cell that holds the definition of where each input data table is to be displayed. There is a row for each input data set which defines the class name, the tag name of the tab to be used, the position and column width definitions (see Section 6.1.2 for further details of format) and the table title.

|  |
| --- |
| function props = getTabProperties(~)  %define the tab and position to display class data tables  %props format: {class name, tab tag name, position, ...  % column width, table title}  props = {...  'MRBreachData','Inputs',[0.9,0.95],{180,60},'Hydraulic data:';...  'Hypsometry','Inputs',[0.35,0.95],{80,420},'Site hypsometry files:';...  'MRSiteData','Inputs',[0.9,0.48],{180,60},'Site data:'};  end |

You may also need to adjust the callback functions used for the menus. In the MRBreach model the highlights illustrate the additions needed to include the Hypsometry menu option under Setup and the use of the InputData class rather than Model class for the input of the model data.

|  |
| --- |
| function MenuDef = inputdataSubMenu(gobj)  MenuDef.MenuLabel = {'Setup'};  MenuDef.MenuList = {'Site Data',’Hydraulic Data,'Hypsometry',...  'Model Constants'};  MenuDef.MenuCallback = repmat({@gobj.inputProps},[1,4]);  end  %%  function inputProps(gobj,src,~)  switch src.Label  case 'Hydraulic Data'  tabname = callClassFunction(gobj,...  'MRBreachData','setInputData',true);  tabsrc = findobj(gobj.GuiTabs,'Tag',tabname);  ModelSummary(gobj,tabsrc);  case 'Site Data'  tabname = callClassFunction(gobj,...  'MRSiteData','setMRSiteData',true);  tabsrc = findobj(gobj.GuiTabs,'Tag',tabname);  ModelSummary(gobj,tabsrc);  case 'Hypsometry'  callClassFunction(gobj,'Hypsometry','setHypsometry');  case 'Model Constants'  gobj.Constants = ConstantData.editConstantData(gobj);  end  end |

[NB – in the call to callClassFunction the green is the class name and the turquoise is the function name. The function accepts two further variables which are flags. The first defines whether the output of the call should be displayed on a tab (eg to display a table of input data and the second flag defines whether the class name should be passed to the function being called. Default values if these variables are not included are [ ] and false, respectively.]

Similarly, it may be necessary to amend the calls for the Run menu options to ensure that the correct functions are called by the UI.

|  |
| --- |
| function MenuDef = runMenu(gobj)  MenuDef.MenuLabel = {'Run'};  MenuDef.MenuList = {'Set Hypsometry','Run Model','Derive Output'};  MenuDef.MenuCallback = repmat({@gobj.runModel},[1,3]);  end  %%  %----Functions called by menu->  %%  function runModel(gobj,src,~)  switch src.Label  case 'Set Hypsometry'  callClassFunction(gobj,'Hypsometry','selectHypsometry');  case 'Run Model'  callClassFunction(gobj,'MRBreachModel','runMRBreachModel');  case 'Derive Output'  gobj.GuiManip = DataManip.getDataManipGui(gobj);  case 'Statistics'  DataStats.getStatsGui(gobj);  end  end |
|  |

## Timeseries data

The InshoreWaves model introduces the handling of timeseries data. There are two new classes, **InWaveData** and **InWaveModel** and they both inherit from the **TSDataSet** class. The **TSDataSet** class includes functions to open one or more text files, load a new timeseries and add data from a file to an existing time series. An instance of **InWaveData** holds timeseries datasets in the property mtsc as a cell array. The property mtsc is initialised by the **TSDataSet** class (mtsc is a tscollection object). The functions in the methods block customise the **TSDataSet** class functions to the specific data handling requirements. To set up a new data class to hold timeseries data requires the inclusion of a standard properties block and a data specific class constructor.

|  |
| --- |
| classdef InWaveData < TSDataSet  %class for offshore wave data  properties (Hidden)  %Data linked to a class data record that needs to be amended if  %the record is deleted.  %Format is: {handle, id used in Cases, other properties}.  %Other than handle, the variable names need to agree with  %variables in the InputFile class, or equivalent  PropertyLink = {'h\_InputFile','idDataSet','DataFile'};  end    methods  function obj = InWaveData(mobj)  %constructor to initialise object  obj.ResDef = mobj.Constants.ResDef;  %initialise list of available input file formats. Format is:  %{'label 1','function name 1';'label 2','function name 2'; etc}  obj.DataFormats = {'Date-Record format','wave\_daterec\_format';...  'Channel Coastal Observatory format','wave\_cco\_format';...  'Cefas Wavenet format','wavenet\_format';...  'ShoreCast data format','wave\_scast\_format'};  end  end  %--------------------------------------------------------------------------  % functions to read data from file and load as a timeseries  % default function for loadData, addData and dataQC are in TSDataSet  %--------------------------------------------------------------------------  end |
|  |

The property DataFormats (defined in TSDataSet) is used to list the different data formats that are to be loaded and the name of the function file that handles each data format (as illustrated by the text highlighted in yellow above).

The data format files have 5 parts:

1. A standard function call that in turn calls one of 3 functions
2. A function to define the meta-data for the data set being imported
3. A function to parse the data ready for adding to the model UI
4. A function called by (iii) to read the data from a file
5. A function to handle data quality control

The timeseries\_format\_template.m file in the Utilities folder provides a template to help when adding new formats.

The **getResDef** function is the same as used in the model function file (see Section 5.2). This defines the ResDef structure property for the class. The fields within the structure define the properties of the variables, xyz data and time to be used when handling the imported data set. The template for the ResDef structure is defined in the ConstantData class and details of the fields within ResDef are detailed in Section 6.8.2.

The **getTSdata** function calls readInputData and then sorts the data into a format that can be loaded with a call to setCollection. There are numerous examples of how to do this in the …/CoastalClasses/FormatFiles folder.

The **readInputData** function is used to extract file header information and then read the data. For files with a single header line and then a fixed number of data columns, the default function, readInputFile can be called. To use this the readInputData function simply defines the data format, as follows:

|  |
| --- |
| function [header,data] = readInputData(obj)  %read wave data (read format is file specific).  dataSpec = '%{dd-MMM-yyyy}D %{HH:mm:ss}D %f %f %d %f %f %f %f %f %f %f';  [header,data] = readInputFile(obj,dataSpec);  end |
|  |

If the file format is defined in the file header, simply set dataSpec = [];

There are several examples and further explanation in Section 5.2 of the InshoreWaves manual.

The **dataQC** function defines any quality control checks to be applied to the data. This function is called from the Setup menu and is not automatically run when the data is loaded. For waves and water levels default QC functions are included (see files: wave\_dataQC.m and wl\_dataQC.m).

When data of given format is loaded it creates a class instance. If more data are loaded (e.g. the same data format for a different location) then this is appended to the same class instance as an additional record. When data of the same type (e.g. wave data) but a different format are loaded this generates a new instance of the class. This allows each data format to have different variables and descriptions whilst all being of the same type. The handling of multiple instances and multiple records is explained in more details in Section 6.8.4.

**InWaveModel** uses ResDef to define how the results are to be saved and plotted. This is defined in the **InWaveModel** constructor block. This class also follows the standard format for a class the inherits from **TSDataSet** and has Transient stsc property that is assigned during the model run and then assigned to the master tscollection property mtsc. This has no real purpose in this model since the results are run in a vector block but could be used to handle each time step and then concatenate the results, if required.

The runInWaveModel function illustrates the use of isValidModel (see Section 6.8.3) and the accessing of a time series that has been loaded for use as input data. The timeseries variables required and any other input variables needed are loaded into the structure ‘inp’ and passed to the external model function hs\_surf.m. The results are then written to a timeseries collection and held as a cell entry in mtsc, as follows:

|  |
| --- |
| %assign the model results to the timeseries collection, tsc  %using the ‘Time’ property triggers the use of a tscollection  tsc = setCollection(obj,[Hsi,Diri,swl,depi],...  'Time',datetime(Time),'metaData',{meta.inputxt});  if isempty(tsc), return; end %trap error before assigning to obj  %get the Case IDs of the input data  h\_wave = getClassHandle(mobj,'InWaveData');  InWaveCID = getCaseID(mobj.Cases,h\_wave,meta.iclass);  h\_wl = getClassHandle(mobj,'WaterLevelData');  InWLevelCID = getCaseID(mobj.Cases,h\_wl,meta.idc);  %--------------------------------------------------------------------------  % End of Model code  %--------------------------------------------------------------------------  %The order of the following calls is important. Results needs  %mobj to have been updated with the new data set  %assign data set to class data property  obj.mtsc{irec} = tsc; %assign to master tscollection for class  %store source CIDs - enables other models to retrieve input data  obj.InputDataCIDs{irec} = [InWaveCID,InWLevelCID];  obj.InshoreBedSlope{irec} = ms; %can be an array or single value  %save pointers to results for the data set loaded  recnum = Results.saveResults(mobj,h\_model,obj);  %add description to the tsc.Name variable  obj.mtsc{irec}.Name = mobj.Cases.CaseDescription{recnum};  ModelUI.myDialog('Run complete');  DrawMap(mobj);  end |

NB: details of the input wave data set that was used to run the model are captured in the property obj.InputDataCIDs and passed to saveResults to be captured in the Case definition.

## Categorical data

The DSDataSet class has functions to load tables from a \*.mat file or a \*.xlsx file.

### Adding a new table

All tables are stored as a separate instance of the class. They therefore have their own meta-data definition in ResDef and can vary to reflect the source data format. The default if to define the default x-axis as ‘C for canonical data in the class object ResDef property xyDefault (see Section 6.8.2 in ModelUI manual).

### Adding a new table class

To add a new table class define a class that inherits DSDataSet. Most of the code needed to load and add data is the available. The properties for the new class define PropertyLink to keep track of the files used to load data. The only method required is the class constructor (Estuary\_db\_data in the example below).

|  |
| --- |
| classdef Estuary\_db\_data < DSDataSet  %class to load estuary database into ModelUI  properties (Hidden)  %Default properties defined in DSDataSet.  %Additional properties:  TableData %cell array of imported tables (could use default ModelData  %defined in DSDataSet)  %Data linked to a class data record that needs to be amended if  %the record is deleted.  %Format is: {handle, id used in Cases, other properties}.  %Other than handle, the variable names need to agree with  %variables in the InputFile class, or equivalent  PropertyLink = {'h\_InputFile','idDataSet','DataFile'};  end  %%  methods  function obj = Estuary\_db\_data(mobj)  %constructor to initialise object  obj.ResDef = mobj.Constants.ResDef;  end  %Other functions as required – eg case specific version of tabPlot or bespoke  %functions for the class  end  end |

Define the default x-axis as ‘C’ in the class object ResDef property xyDefault (see Section 6.8.2 in ModelUI manual) if the row names extracted from the data being loaded are character strings and can be used as canonical data.

## Multi-dimensional variables

Where a variable is not a vector in time or space, but multi-dimensional, this is handled by defining the space dimensions and time as properties of the variable(s). This uses definitions in the model ResDef property to provide details of the XYZ dimensions using ResDef.xyz\*\*\*\* and details of the time dimension using ResDef.row\*\*\*\* (see Section 6.8.1 for more details).

For example, in Diffusion2D, the DiffusionModel loads 2D or 3D XYZ data and time. This requires the various variables to be defined in ResDef as part of the constructor for the DiffusionModel class. The saving of the data is then handled by the call to setCollection, where the model ‘results’ are passed as a single cell array. If there were more variables these would be in additional cells of the ‘results’ array. The definition of XYZ is passed as ‘xycoords’, which is a cell array holding a vector for each dimension. Time is passed in ‘modeltime’ using the keyword ‘rowNames’. The code handles Date-Time and Duration formats. However, for this to be handled properly the correct format must be defined in the ResDef variable rowFormat (e.g as {'dd-MMM-uuuu HH:mm:ss.SSS'} in the DiffusionModel class).

|  |
| --- |
| [results,xycoords,modtime] = diffusion2Dmodel(inp,run);  %now assign results to object properties  startyear = sprintf('01-01-%d 00:00:00',0);  modeldate = datetime(startyear,'InputFormat',obj.ResDef.rowFormat{1});  modeltime = modeldate + seconds(modtime);  %each variable should be an array in the 'results' cell array  %model returns single variable as array of doubles: so use {results}  newds = DataSet.setCollection(obj,{results},...  'xyzData', xycoords,'rowNames',modeltime);  if isempty(newds) return; end %trap error before assigning to obj  obj.ModelData{irec} = newds; |

Note the requirement for a timeseries collection is that all variables in the collection are the same length with the same time vector. They can be saved to a timeseries collection using the ‘Time’ keyword. If the model output is multi-dimensional rather than a vector and includes time, a Matlab table is easier to handle and is saved using the ‘rowNames’ keyword. Some further details of how data can be stored using the setCollection function are given in Section 6.8.1.

## Altering the Plotting UI

The default plotting UI within ModelUI has three tabs defined for XY, XYZ and Animate plotting options. The inclusion of one of more these tabs is defined using the NumTabs property in the ModelUI class (or its equivalent class in a new model). These implement the following options:

XY: data selection for a 2D plot, where the default X variable is defined by ResDef.xyDefault.

XYZ: data selection for either a 2 or 3D plot.

Animation: data selection for an animation (requires some form of time series data).

The layout of these tabs is defined in the **ModelPlots** class, with a standard template used for each tab (naming format is ‘set<tabname>tab’). **ModelPlots** can also be used to control the behaviour of the UI and includes functions to get the data for the selected options. The tab buttons New, Add and Delete call the getPlot function from the **PlotFig** class. This class then generates the figure and creates the plot. If only the figure layouts need modifying this can be done by overloading **PlotFig** class, whereas if a different UI is needed for data selection then **ModelPlots** may need to be modified/overloaded.

## Additional programming notes

### Data assignment

There are two options for saving data sets within ModelUI. For timeseries data with a date and time the Matlab tscollection class can be used. For vector and array data, the Matlab table class should be used. The structure of the data and metadata in both are similar and provide for:

1. time, or an index;
2. positional information for the data points (x, y, z), where these are fixed in time;
3. variables that match the time/index and co-ordinate dimensions (e.g. a timeseries of tidal currents on a 2D grid would have t, x, y and a 2D array of each variable at each time step).

NB: only vector data that matches the time vector can be saved as a variable in a timeseries collection. Multidimensional variables are therefore best handled using the table option.

When assigning an imported data set, or model results as a case/scenario within ModelUI the data format and metadata options are as detailed in Table 1.

Table 1 -timeseries and matrix array data assignment within ModelUI

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | ResDef | Timeseries | Matrix array |
|  |  | tscollection. | table.Properties. |
| Time or index | rowName | Time | RowNames |
| XYZ positions |  | TimeInfo.UserData | UserData.XYZ |
| Case description |  | Name | Description |
|  |  | For each timeseries: | For each variable array: |
| Data set |  | Data | *table column* |
| Variable name | varNames | Name | VariableNames |
| Description of variable | varDesc | DataInfo.UserData | VariableDescriptions |
| Units of variable | varUnits | DataInfo.Units | VariableUnits |
| Variable Labels | varLabels | UserData.Labels | UserData.Labels |
| Metadata | - | UserData.MetaData | UserData.MetaData |
| Class name | - | UserData.Class | UserData.Class |
| Data type (model, raw) | Type | - | - |
| Quality information |  | QualityInfo.Code | n/a |
| Quality description |  | QualityInfo.Description | n/a |

Grey text: use not used in setCollection

var\* assigned to timeseries or table in setCollection. This allows variable definitions to vary for each table or timeseries even without the need to change the class definition (ResDef).

Variable Description provides a description of a variable and are used to define the plot legend. Variable Labels provide a collective definition to group variables and are used to label the plot axes. See Vertical Profile model for an example of this usage.

Class name is used in the Export/Import function to assign data to correct class if it exists

The assignment of the data within the setCollection function makes use of definitions provided in the ResDef property that is defined as part of the model constructor (see Section 6.8.2).

|  |
| --- |
| %demo code for assigning time in a model  %nint = number of time intervals  %time\_variable = time in required units or datetime format    %case 1 – time from time = 0  startyear = sprintf('01-01-%d 00:00:00',0);  modeldate = datetime(startyear,'InputFormat',obj.ResDef.rowFormat{1});  modeltime = modeldate + seconds(runtimedata);  %the input format is defined in the model ResDef.rowFormat property.  %e.g. in the DiffusionModel class:  ResDef.rowFormat = {'dd-MM-uuuu HH:mm:ss.SSS'}; %using MM avoids problems with ‘Locale’  %and the output format is defined in ResDef.rowUnits property  r.rowUnits = {'s'}; %used to identify units  %in this example the data is stored in Date-Time format and output as durations in seconds. Permissible output formats include any valid Matlab Data-Time or duration format. |

The function setCollection is in the DataSet class interface.

The model results or imported data are passed to the function in the variable ‘VarData’.

Rows in timeseries or table data are time

*For timeseries data*

* time is defined in variable 'Time';
* xzy data is defined in variable ‘xyzData’ and saved in TimeInfo.UserData as a cell array;
* variables are vectors with the same length as time.

*For table data*

* time is defined in variable 'rowNames';
* xzy data is defined in variable ‘xyzData’ and saved in Properties.UserData.XYZ as a cell array;
* variables can be multi-dimensional arrays and each cell array is in x,y,z order.

### Defining the model output properties

The formats for loading data using the function setCollection and the way in which data is handled in the output utilities, such as plotting and editing, requires information about the model output, or the data set being loaded (see classes **ReferenceData** and **InWaveData**) to be provided in the property ResDef. A default structure for ResDef is given in the ConstantData class and can be used to initialise the definition of ResDef. Not all variables in the structure need to be defined for a given model. However, as a minimum the DataVar, var\*\*\*\* and either the xyz\*\*\*\* or row\*\*\*\* variables (both if there is time and xyz data) and xyDefault, Type and Style. The full set of variables are as follows:

|  |  |
| --- | --- |
| ResDef variable | Description |
| DataVar | property name for model output array |
| AdnOutVar | property names for additional outputs |
| InCase | when a Case is used as an input to a model, pass caseid to Results |
| varNames | names used in tscollection/table to label variables |
| varDesc | description of variables used in data access UIs and plot legends |
| varUnits | variable units |
| varLabels | axis labels for results (e.g. a collective name for variable descriptions) |
| xyzDesc | description to be used for x, y and z co-ordinates |
| xyzUnits | units for the defined co-ordinates |
| xyzLabels | axis labels for use with XYZ data |
| rowDesc | description for RowNames in table (usually Time but rows can be any unique descriptor) |
| rowUnits | units for the data in RowNames. Formats can be durations: y,d,m,s or datetime: dd-MMM-uuuu HH:mm:ss |
| rowLabels | axis labels for use with row data |
| rowFormat | time format to use when saving time data (this is used for both table and tscollection classes) |
| xyDefault | Defines variable to use for the x-axis in XY plots ('Time' or 'X', 'Y',’Z', or ‘C’) |
| Type | used to write results to Excel spreadsheet: 'Single'-all variables on one sheet'; ‘Multiple'-Xdata and one variable per spreadsheet |
| Style | data type (‘model’, ‘data’, etc for partitioning tab display) and some functions in plotting UI. |

### Checking for a valid model

The function validModel, which can be found in GUIinterface, provides a simple way of checking the model inputs. The property inputHandles defines the class handles that are needed to run the model and is set in the call to setGUI in the UI constructor. The function validModel checks first to see if the required model handles have been instantiated. validModel then checks that each valid handle has entries for all fields needed to run the model. This tests all those properties that are exposed (not Hidden) in the class and assumes that the input data class inherits from PropertyInterface.

### Data structure

Each case saved within ModelUI holds a data set which may contain timet,t , space dimensions that are fixed (e.g. in time) for the variable set[[3]](#footnote-3), xyz, any number of variables, v(n), and an output definition ResDef. ResDef maps onto t, xyz and v(n). Within a single derived class instance, all definitions are constant (ResDef) but t, xyz can vary for each Case that is added.

 Cases for a model or data added are held in the' Cases' handle. For each case added, the Results class is called, a unique caseid is assigned and access details for the class being created are saved. This includes the CaseId, CaseHandles and CaseHandleID.

From a list of cases (Cases) a selection returns the case to be used (useCase) from which the CaseID is given by:

> caseid = mobj.Cases.CaseID(useCase);

This is done because the user may delete records. If the cases can be created by multiple classes, it becomes necessary to find them using the unique CaseID.

CaseID is used to find the individual record using:

> [h\_rec, iclass, dprop, id\_rec, aprop] = getCaseRecord(obj, mobj, caseid);

where the call includes obj, the Cases handle, mobj, the ModelUI handle, and caseid, as above. The function returns the handle for the class being retrieved, h\_rec, the id of instance of the class, iclass, the list of data properties, dprop, the record number of the case, id\_rec, and the list of any additional properties that may have been saved, aprop. This can then be used to retrieve a data set as follow:

> dataset = mobj.(h\_class)(iclass).(dataprop{useProp}){id\_rec};

which selects from multiple instances (iclass) of a class (h\_class) and/or multiple instances {id\_rec} of a property (dataprop{useProp}), where useProp is the id of the property that holds the data within ResDef.DataVar. In the models implemented for ModelUI, ResDef.DataVar={'ModelData'}; so that useProp=1.

An individual variable is then obtained using:

> datavar = dataset{:, useVar};

where useVar is the selected variable as given in DataSelection.(χ){2,2} (see Section 6.8.5 and 6.8.7 for details).

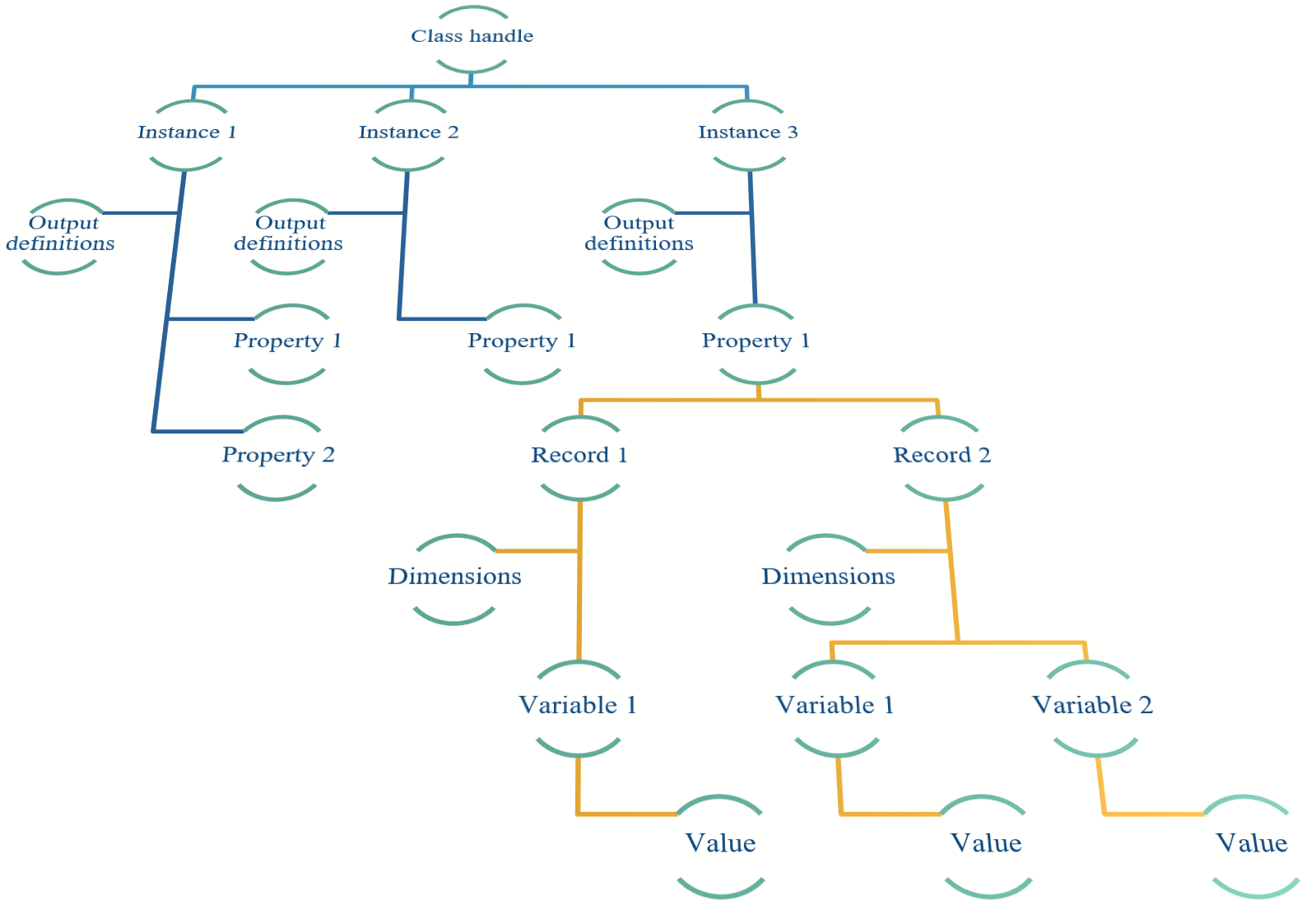
The following three functions provide direct access to a case handle, a dataset and a variable

casehandle = getCaseHandle(obj, mobj, useCase)

[casedataset, casehandle] = getCaseDataSet(obj, mobj, useCase)

casevardata = getCaseVariable(obj, mobj, useCase, useVar) (works for tscollection and table)

The following figure illustrates the generic data structure. For an example of its implementation for the Derived class see Section 6.8.6.



Within this structure it is possible to hold:

1. multiple instances of a class, each with their own output data definition (ResDef) comprising variables, xyz dimensions and row (time) dimension.
2. multiple properties for each instance (e.g. ModelData and OtherData)
3. multiple records (values, tables, or tscollection) of each property, each with their own dimension definition (time and xyz) and any number of variables
4. variables with up to 4 dimensions labelled time and x, y, z but in principle can be any controlling dimension (e.g. time is simply the label given to table rows but this can be any variable that is unique (no duplicates)).

### Selections on tabs in DataGuiInterface

The order in which the tabs are displayed is independent of the order in which they are saved in the DataSelection structure for each variable (X, Y, Z and current selection C - denoted collectively as χ)[[4]](#footnote-4). This is done so that tabs can be created to suit applications, with selection controls in any order, but the definition of all variables selected (case, variable, scaling, limits, etc) is in a defined structure to simplify the retrieval of records in different applications.

The order of the Selection UIcontrols on a tab is defined in the cell array TabContent.Order. These must be unique identifiers and the default set defined in DataGUIinterface is:

DSorder = {'Case', 'Variable', 'Scaling', 'Limit1', 'Limit2', 'Limit3', 'Limit4', 'Limit5',’Type’, 'Other'};

These are inititalised in setVariableList. A change made by the user to a single Selection UIcontrol ('sel\_uic') is handled in updateSelection, with application specific changes defined in ControlSelection. These two functions also maintain DataSelection.C to match the current selection in the Selection UIcontrols (see also Section 6.8.7).

To access a selection one can use the index of the variable required, or use find to get the index, e.g.:

idx = find(strcmp(gobj.DSorder, 'Type'));

type = gobj.DataSelection.C{idx, n};

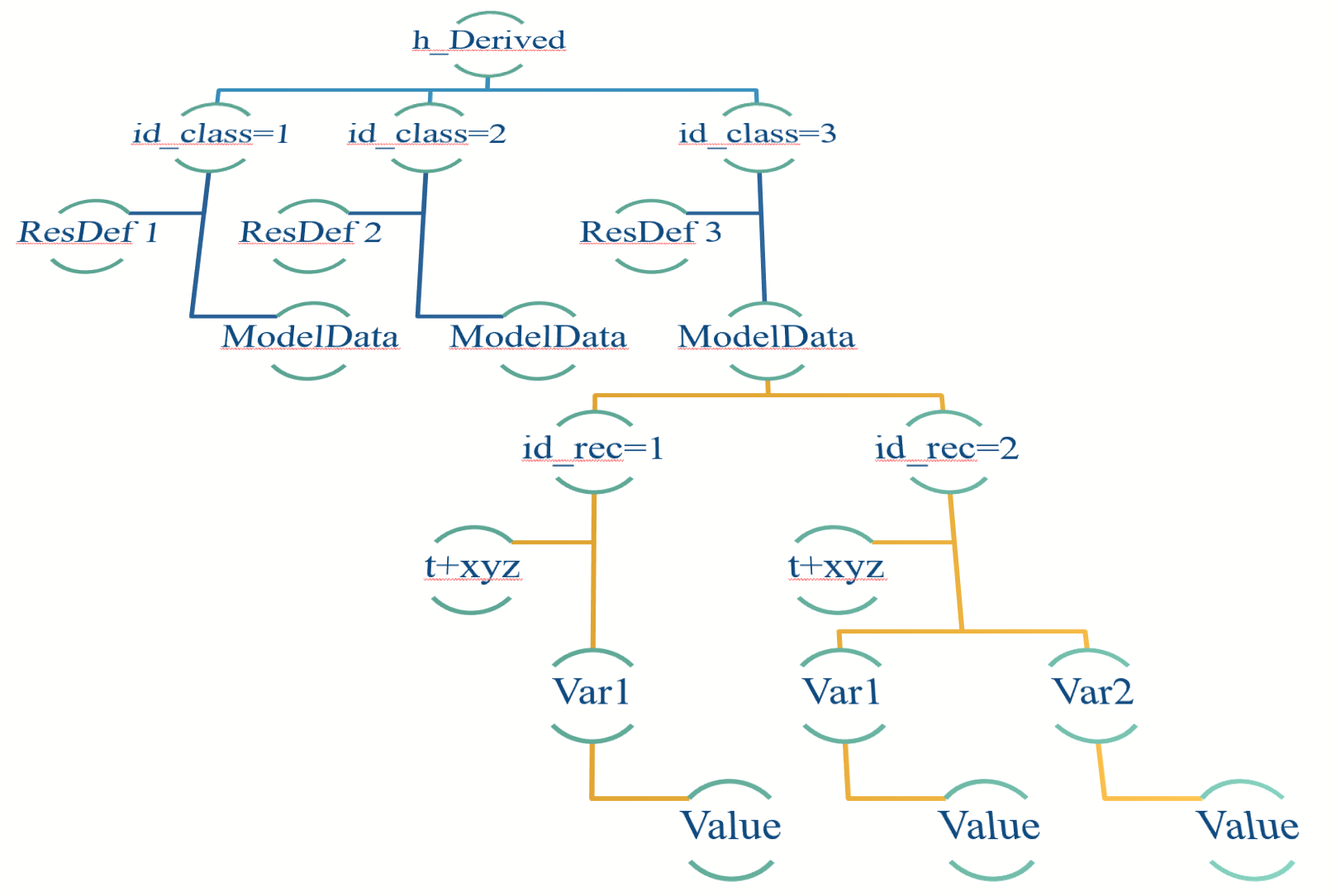
The value of ‘n’ is 1 to return the text selection or 2 to return the numeric value.

### Permissible ways of adding data to a class handle or property

The models that are implemented as part of the ModelUI package all use the same format for holding data and accessing data. In each model or data class, a variable is defined to hold the data set. A single instance of the model (or data) class is created with an associated set of Result Definitions (obj.ResDef). New cases are then added to the property that holds the data to create a cell array (e.g the ith record, irec, would be: obj.ModelData{irec}). This is done because the variables to be saved from a model, or imported data set, are usually constant. They are assigned to a cell array so that the size of the data set can vary from one case to another.

The Run>Derive Output option (see Section 3.5) implements two other methods.

When deriving output, you may wish to create a completely new set of definitions for a variable, or once you have created at least one derived case, to add subsequent cases to the existing definition (same as for models), or if the dimensions of the derived output are the same as previous outputs, to add the new variable to an existing case (see Section 4.6). All three of these options are possible and the data structure to do this is illustrated below. There can be multiple instance of the class (iclass 1, 2, 3, …) referenced from the class handle (h\_Derived). Each instance has a ModelData property which can have multiple records (id\_rec 1, 2, 3, …). Each record is a table that has defined dimensions (t, x, y, z) and as many variables as required. The variables must all have the same number of rows but can be a vector of values, or multi-dimensional arrays (e.g. xyz), or some combination of the two.



To add a case as a new instance of the class, rather than a new record in an existing instance, the new instance, obj, is assigned to the class handle, incremented by iclass = length(h\_Derived)+1. In addition, the value iclass is passed as an additional variable in the call to saveResults. If iclass is omitted, saveResults assigns a value of 1.

mobj.h\_Derived(iclass) = obj;

Results.saveResults(mobj, 'h\_Model', obj, iclass);

To add a case as new, when there are multiple instances of the class, iclass will be the instance to which you want to add the case and irec = length(ModelData)+1 for chosen instance. The calls are similar:

mobj.h\_Derived(iclass).ModelData{irec} = obj;

Results.saveResults(mobj, 'h\_Model', obj, iclass);

To add as an additional variable to an existing case find the case (see Section 6.8.4) and then use either addTSvariable, or addDSvariable to add to a tscollection or table respectively.

### Data retrieval in classes that use DataGuiInterface

Once UseData and getData have been called, the variable selections are assigned to DataSelection.X, DataSelection.Y, DataSelection.Z. These cell arrays have the selections as text in the first column and the integer values of the Selection Value (id of position in list) in the second column. The row order is {'Case','Variable','Scaling','Limit1','Limit2','Limit3','Limit4','Limit5',’Type’,'Other'}

Selecting a case from the Case list gives the id to the record in Cases.CaseID. However, because this list comprises cases from many different classes and records can be deleted from this list, this value does not uniquely identify the record. The caseid = obj.Cases.CaseID is therefore used to retrieve the unique record identifier. This can then be used in getCaseRecord to recover the handle, case and the list of variables for the selected case, from the array of datasets for that class. These variables can then be used to recover a specific record, eg: to select the first variable

dataset = mobj.(h\_class)(iclass).(dataprop{useProp}){id\_rec}; where useProp is the selected variable as given in DataSelection.(χ){2,2}. See Section 6.8.6 for details of how to use multiple instances (iclass) of a class (h\_class) and/or multiple instances {id\_rec} of a property (dataprop{useProp}).

### Supressing the Scenario prompt

When data is loaded, or a model run is completed the user is prompted to provide a “Scenario description”. To supress this prompt simply add a ‘casename’ when saving the results:

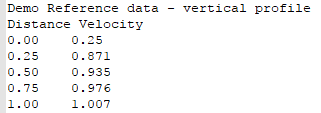
Results.saveResults(mobj, handle, obj, iclass, casename);

Where ‘mobj’ is the modului handle, ‘handle’ is the class handle for the results being saved, ‘obj’ is the instance of the class, ‘iclass’ is the index for the record in the class object and if included, ‘casename’ is the name used to identify the record (and displayed on the Cases tab).

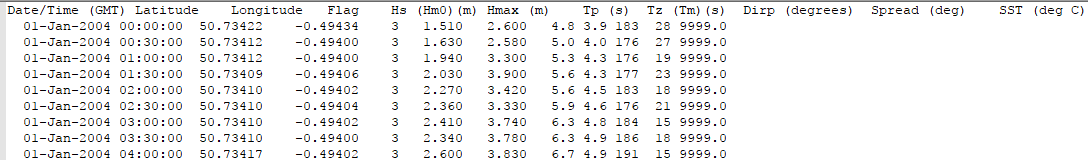
### Input file formats

The following are examples of file formats used. For details of options for a wider range of time series data (wind, waves, water levels, etc) see the InshoreWaves manual.

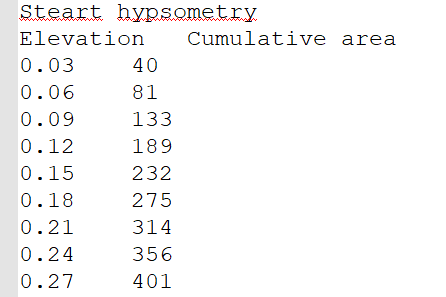
ReferenceData loads text files with the following format



InWaveData loads text files with the following format (standard CCO wave data format)



Hypsometry loads text files with the following format



# Program Structure

The overall structure of the code is illustrated schematically in *Figure 2*. This is implemented through several classes that handle the graphical user interface and program workflows (Main GUI) and several classes that handle the data manipulation and plotting (Data GUIs). The interfaces and default functionality are implemented in the ModelUI package using the following classes:

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | *Figure 2 – High level schematic of program structure* | |  | | ***GUIinterface*** – basic functionality of Main GUI  **ModelUI** – implements GUIinterface for the ModelUI application  ***DataGUIinterface*** – basic functionality for Data selection  **ModelPlots** – implements DataGUIinterface to plot results  **DataEdit** – implements DataGUIinterface for editing data sets  **DataManip** – implements DataGUIinterface to derive new data sets from existing ones  **DataSelect** - implements DataGUIinterface for data selection in stand-alone functions  **DataStats** – implements DataGUIinterface to provide a range of statistical analysis functions  ***PropertyInterface*** – basic functionality for handling data input and display  ***DataSet*** - basic functionality for handling timeseries data  ***DSDataSet*** – additional functions to import and handle table datasets. Inherits DataSet.  ***TSDataSet*** – additional functions to import and handle timeseries datasets. Inherits DataSet.  **ModelSpecification** – handle model specifications for different models for default ModelUI  **Project** – details of project definition (name and date) and path and file for project  **ConstantData** – defines constants used in model  **Results** – store model run details as Cases and handle saving Cases to an excel file.  **PlotFig** – defines a range of different plot types  **InputFile** – handles the input file list for reference data and imported timeseries data  **RunModel** – template for models that require extensive pre and post processing as part of the run  **NumInputdlg -** Modifies inputdlg Matlab function to handle numeric input and output  **Tabledlg** - Defines a table with rows and columns with the option to edit variables defined in the call to Tabledlg. | |

In addition, ModelUI uses the following class:

**Derived** – class used to hold the results generated by the Derive Output UI, DataManip.

**Model** – defines model output properties and implements model functions.

The function polarplot3d.m is used by **PlotFig**.

# Bibliography

Bartlett, R., Mortimer, R.J.G., Morris, K., 2008. Anoxic nitrification: Evidence from Humber Estuary sediments (UK). Chemical Geology, 250(1-4), 29-39.

Coles, S., 2001. An Introduction to Statistical Modeling of Extreme Values. Springer Series in Statistics. Springer-Verlag, London.

Prandle, D., 1982. The vertical structure of tidal currents and other oscillatory flows. Continental Shelf Research, 1(2), 191-207.

Taylor, K.E., 2001. Summarizing multiple aspects of model performance in a single diagram. Journal of Geophysical Research - Atmospheres, 106(D7), 7183-7192.

Townend, I.H., 2008a. Breach design for managed realignment sites. Proc.Instn Civ.Engrs., Maritime Engineering, 161(MA1), 9-21.

Townend, I.H., 2008b. Hypsometry of estuaries, creeks and breached sea wall sites. Proc.Instn Civ.Engrs., Maritime Engineering, 161(MA1), 23-32.

# Appendix A – DataGuiInterface function calls

Call sequence to initialise ModelPlots using the DataGuiInterface:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| mp.getPlotGui |  |  |  | *Callbacks* |  |
|  | mp.PlotGui | (*class constructor*) |  |  |  |
|  | setDataGuiFigure |  |  |  |  |
|  | setDataGuiTabs |  |  | setTabActions |  |
|  | *for each tab*: | setTabContent |  |  |  |
|  |  | setVariableLists |  |  |  |
|  |  | setDataOptionControls |  | updateSelection |  |
|  |  |  | setSlider |  | ControlSelection |
|  |  | setXYZpanel |  |  |  |
|  |  |  | captureButton | XYZselection |  |
|  |  |  |  |  | XYZbuttonOptions |
|  |  |  |  |  |  |
|  |  |  | captureTextBox |  |  |
|  |  | setAdditionalButtons |  | *defined in setup* |  |
|  |  | setTabControlButtons |  |  |  |
|  |  | *depends on button* | mp.UseData |  |  |
|  |  |  |  | PlotFig.getPlot |  |
|  |  |  | ExitDataGui |  |  |
|  |  | initialiseXYZoptions |  |  |  |
|  |  |  | resetXYZoptions |  |  |
|  |  |  | updateSelection |  | ControlSelection |
| In addition PlotFig | calls: |  |  |  |  |
| PlotFig.getPlot |  |  |  |  |  |
|  | getData |  |  |  |  |

# Appendix B – Installation using zip file

The ModelUIcore.zip file contains a series of sub-folders and the demonstration models are supplied in the ModelUIdemo.zip file. These should be extracted and placed in a common folder, for example, suing a top level folder entitled MUImodels:

MUImodels > CoastalFunctions (functions used in some models)

> Diffusion (model code)

> InshoreWaves (model code)

> ModelUI (model interface code)

> MRBreach (model code)

> SimpleTide (model code)

> Utilities (useful scripts and templates)

> VerticalProfile (model code)

Unzip the file to wherever you want to setup the working folders. There is a ~Readme file in the top level folder that details the folder structure. Each model includes a ‘run\_\_<modelname>.m’ script file, which initialises the paths required for that model and then calls the model and needs to be run with the specific model as the working folder in Matlab. For ModelUI itself the function call is:

>> run\_\_modelui

If you plan to run other models and move folders while working, there is a utility in the Utilities sub-folder called addModelUIpaths.m[[5]](#footnote-5). This sets the required paths for all ModelUI models. These are removed at the end of each MatlabTM session, or can be removed using the utility delModelUIpaths.m, which is also in the Utilities sub-folder. To make these permanent path changes, see “Add Folders to the MATLAB Search Path at Startup” in the MatlabTM help documentation.

Once the folder paths have been set-up, type ModelUI; in the command window to run the model. This initialises the interface and the user is prompted to select a model from the list of models currently defined within ModelUI. Alternatively, the following command loads a specific model:

*>>ModelUI(‘<model name>’);* [Where model name is a defined model, such as ‘Vertical Profile’, or ‘Simple Tide’]

Note: By installing the MUImodel toolbox, the paths are defined when the toolbox is installed using ‘Add-Ons’ from the Home tab in Matlab™.

1. Various pre-defined function templates can be accessed using the ‘Function’ button. Alternatively, text can be pasted into the equation box from the clipboard by right clicking in the text box with the mouse. [↑](#footnote-ref-1)
2. If there are no changes required to the main user interface (such as different menu or tab options) then a bespoke model can simply be added to ModelUI, as explained in Section 5. [↑](#footnote-ref-2)
3. Some data sets have space dimensions that vary at each time step – such as beach profiles where the location of the measurements changes for each survey. For such cases the special co-ordinates need to be stored as variables for each time interval. [↑](#footnote-ref-3)
4. X, Y, Z are the selected variables 1 to 3 in the GUI. They are not spatial dimensions unless assigned as such. [↑](#footnote-ref-4)
5. When new models are added as sub-folders to the MUImodels folder, these are also added to the path when addModelUIpaths is run. [↑](#footnote-ref-5)