

iFlow modelling framework

User manual & technical description

Yoeri Dijkstra

Copyright © 2017. Y.M. Dijkstra

When using iFlow, please cite Dijkstra, Y. M., Brouwer, R. L., Schuttelaars, H. M., and Schramkowski, G. P (Manuscript submitted to Geoscientific Model Development). The iFlow Modelling Framework v2.4. A modular idealised process-based model for flow and transport in estuaries.

Additionally you may refer to this manual as Dijkstra, Y. M. (2017). *iFlow modelling framework. User manual & technical description*.

Note the license obligations that come with iFlow.



Contents

1	Introduction	5
1.1	What is iFlow	5
1.2	Overview of the some important elements	5
1.3	Terms of use	7
1.4	How to read this manual	7
2	Getting started	9
2.1	Installation and requirements	9
2.2	Running a simulation	9
3	Starting to use iFlow	11
3.1	Building and using modules	11
3.2	Input files	15
3.3	Variable names and sources	16
3.4	Dimensions and grids	17
3.5	Accessing data	21
3.6	Specifying analytical functions	26
3.7	Shape of array data	28
3.8	Output storage	29
4	Advanced features	31
4.1	Numerical functions	31
4.2	Iterative modules	33
4.3	Dynamic options for registries	34
4.4	The NiFTy toolbox	36

4.5	The sTeP toolbox	42
A	Object-oriented programming	47



1. Introduction

1.1 What is iFlow

The iFlow Modelling Framework (in short: iFlow) is an extendible, modular modelling framework. As such, the framework provides tools that support and simplify the development of a scientific model. It specialises in supporting modular models; i.e. models that can be subdivided in different tasks, or modules. These modules may interact with one-another in a sequential or iterative way.

iFlow facilitates the model development by arranging the interaction between modules, by arranging input and output, and by providing tools for common scientific operations, such as numerical differentiation, integration and interpolation. The modelling framework additionally supports collaborative model development. This is because the modular structure is defined in such a way that one developer/user can use a module written by someone else, without having to bother about the way this module is coded. Modules written by different programmers can thus easily interact in a single program.

One of the unique features of iFlow is the seamless integration of both numerical and analytical model components. This means that one can choose to implement a task using either numerical mathematical methods or analytical functions, but access the computed data in the same way. This makes the modelling framework especially suitable for idealised models, in which certain tasks are simplified to such an extent that they can still be performed using analytical formulations, while other tasks need to be solved numerically. The analytical functions bring accuracy and speed to the model, while the numerical methods allow for flexibility and a wide range of applications.

1.2 Overview of the some important elements

Before explaining the iFlow in all its details in the next chapters, we will shortly explore some main elements of iFlow here. We will skip over many important features and omit any specific instructions. Instead, this section is meant to familiarise one with some of the basic concepts of iFlow.

The iFlow program is programmed in Python using Object-Oriented Programming (OOP, see Appendix A for a short introduction). We will often refer to basic OOP related terms such as classes, methods and inheritance in this manual. We will also show some examples of Python code throughout this manual, which can likely be understood using some basic knowledge of notation and data-types in python and experience in any other programming language.

iFlow can be subdivided into two main elements: the modelling framework and the packages of modules and functions. The modelling framework is that part of iFlow that arranges all overhead, input, output requirements, access to data and variables. The framework additionally provides some scientific operations, such as numerical differentiation, integration and interpolation. In essence the modelling framework is an 'empty shell', i.e. it does not contain any content. The content is provided in the form of packages of modules and functions that can be developed for any scientific modelling application.

Modules are the basic building blocks of a model in iFlow. A module is essentially a class with the additional requirement that it contains a (public) method called 'run' and that takes no arguments. Modules get their input through an argument in the init method. The output of a module should be provided as a single dictionary that is returned in the run method. Within a module, one can use any additional methods, use other classes or functions, access files etcetera. Modules should however not call or interfere with other modules.

Modules should be included in packages (i.e. folders with an empty file called `__init__.py`) which can be placed at any location on the local machine or on a network disk. A package can include multiple modules, sub-packages, functions, classes or other files. A top-level package (i.e. not sub-packages) containing modules is required to contain a file called 'register.txt'. In this file one should register the modules in the package and specify the input requirements and output variables of each module. Any other class, function or file in the package does not need to be registered.

The interaction between modules works through a system of shared variable names; i.e. each variable is assigned a string name that it keeps throughout the program. Using this string name, a module can access the variables that are specified for it in the input file and the variables specified by modules run earlier. As a consequence, a collaborative project requires the different authors to agree on the names of the variables used in their modules. A variable can take a number of data types. It can be a scalar, string, list, numpy array or reference to a (possibly analytical) function description. Whatever the type of the variable, the way it is accessed has been made uniform through the wrapper class 'DataContainer'. This wrapper class provides methods to access any variable by its string name and any other relevant arguments. The DataContainer is an extremely powerful feature of iFlow. If for example the variable accessed is a numerical array, the DataContainer will return the requested data points and will automatically interpolate the numerical data if necessary. If on the other hand the variable is a reference to an analytical function, the DataContainer will evaluate this function on the requested data points. The DataContainer can also be used to access derived quantities of both numerical and analytical data, such as the derivative of a variable. It will then automatically search if an analytical derivative is specified and will otherwise compute the numerical derivative.

The modelling framework coordinates the data management and the running of modules. From the input file, the framework will make an inventory of the modules specified. It will then use the registry of the module to check if all input variables are provided. Consequently it will use the registry to determine the order in which the modules should be run and whether an iterative procedure might be required. It can even decide to completely ignore a module if it is not necessary. After running a module, the modelling framework will make sure that the returned data is added to a DataContainer so that becomes available

for all other modules. At the end, the framework makes sure that the output is saved to a specified output folder.

1.3 Terms of use

When using iFlow in any scientific publication, technical report or otherwise formal writing, authors are strongly requested to cite

Dijkstra, Y. M., Brouwer, R. L., Schuttelaars, H. M., and Schramkowski, G. P. (Manuscript submitted to Geoscientific Model Development). The iFlow Modelling Framework v2.4. A modular idealised process-based model for flow and transport in estuaries

The iFlow code is property of the Flemish Dutch Scheldt Committee (VNSC) and is licensed under LGPL (GNU Lesser General Public License). In summary, this means that the code is open source and may be used freely for non-commercial and commercial purposes. Any alterations to the iFlow source code (core and modules) must be licensed under LGPL as well. However, new modules or a coupling between iFlow and other software may be published under a different licence. Nevertheless users of iFlow are encouraged to make their own developed modules and model applications open source as well.

1.4 How to read this manual

The aim of this manual is twofold: it functions as study material to learn how to work with iFlow, but is also a reference document useful for looking-up details while programming. We have tried to suit both goals by providing textual explanations as well as lists, tables and code samples for quick reference. Throughout the first chapters in this manual we take the viewpoint of a user and model developer. We will explain the features of iFlow and explain how to use them. We advise first users to start by reading these chapters, as the background knowledge is absolutely necessary in order to operate iFlow and develop modules for it.

This manual is structured as follows: Chapters 3 and 4 take a model user/developer viewpoint. Chapter 3 treats the basic building blocks of iFlow, such as modules, input, output, access of variables, analytical functions and numerical grids. These are the elements that are guaranteed to be relevant to the development and use of a model and we strongly advise first users to read this chapter. Chapter 4 discusses some more advanced features that might not be required in a first program. However, the features treated here are not too exotic and it is likely that a somewhat more involved project uses these. We therefore recommend to scan this chapter and read sections when they become relevant to your project. Finally, Chapter ?? presents the internal structure of the modelling framework. This chapter is not directly relevant to users/developers of models, but is important when one wants to make changes to the modelling framework itself.



2. Getting started

2.1 Installation and requirements

The iFlow modelling framework is written in Python, which is a free, open source programming language. In order to run iFlow you need to install Python 2.7 and a number of additional packages that enhance Python's functionality. We recommend to use one of the many freely available a pre-built Python bundles, which combine Python with the most commonly used packages. We provide our experience with some tested packages below

Python bundle	Notes
Anaconda 2.4.1	Recommended bundle
Anaconda 4.0.0	Newer version than Anaconda 2.4.1, but has problems in combination with loading exported files from Matlab
Canopy 1.5.2	Similar to Anaconda, but uses a different version of the plotting packages, which is not compatible with the standard plotting tools in iFlow.

These packages have been tested on all platforms.

In addition, the iFlow package `semi_analytical` Brouwer (2017) requires the package `scikits.bvp_solver`, which requires a Fortran compiler.

iFlow can be run using either the command line and any code editing program or any IDE. An IDE provides an integrated environment for code development, running and debugging and is therefore recommended. We recommend using PyCharm, which is a free IDE especially developed for Python programs.

2.2 Running a simulation

To start using iFlow we recommend to do our tutorials, which are available with the source code. These provide a step-by-step introduction to using the most important features of iFlow and are focussed on those who want to use iFlow, not develop new modules. This

manual treats all features of iFlow, but takes a more detailed approach suitable for iFlow users and developers.

To run iFlow, start the command line or the python IDE. Next run the file `main.py` in the iFlow root directory. To do this from the command line, browse to the iFlow root directory and type `python main.py`. When using an IDE, please consult tutorials for your IDE to run the program. iFlow will display its main menu, such as in Code sample 2.1.

■ **Code sample 2.1 — iFlow main menu.**

```

1 '''
2 iFlow version 2.3 beta
3
4 No working directory set. Now working from the iFlow root directory.
5 Enter cwd to change the working directory.
6
7 Please choose an input file from the list of recent files:
8     1  input/base_model.txt
9     2  input/advanced_model.txt
10    3  input/expert_model.txt
11 or enter the path to a new input file:
12 '''

```

■

After the version number on the first line, the next line displays the current working directory. If no working directory is set, iFlow will work from its root directory. It is recommended to work from a directory other than the iFlow root directory. The working directory is the standard directory from which iFlow runs its input files and where it writes its output files. The working directory can also be used for additional project-specific scripts, e.g. for plotting results. To change the working directory, type `cwd`, press enter and type the absolute path to the working directory. working directory

iFlow works with input provided in a text file. The next lines in the menu allow you to select an input file. The numbered items show the maximum five most recently used files. To select either one of these, type the number in front of the file name and press enter. To run the most recently used file, one can alternatively press enter without typing anything. A different input file can be entered by typing either the absolute path to the file or the path relative to the working directory. The file name should be entered including the extension. input file

3. Starting to use iFlow

3.1 Building and using modules

What is a module

Modules are the basic building blocks of an iFlow model. A module is particular task of a model that can be performed relatively independently. That is, a task that may possibly take input from other tasks and produce output needed for other tasks, but is otherwise independent. A set of modules may be executed in a single sequence or in a loop. At this stage we will focus on single sequences of modules. A loop over modules can be forced by implementing iterative modules, which will be treated in Section 4.2.

The design of the module structure is the first task that a model developer should focus on. It is strongly advised to define modules as substantial tasks that make immediate sense to anyone that is somewhat involved in the model. In other words, a module should represent the calculation of one or more variables, while stowing the details of the implementation away inside the module.

Basic module structure

A module is a python class, the minimal form of which is shown in Code sample 3.1.

■ Code sample 3.1 — Minimal form of a module.

```
1 """
2 MyModule
3 In this location, write comments that make sense to anyone that wants to use this module.
4 Here we specify a dummy module that returns a variable 'pi' which is a scalar
5
6 Date:
7 Authors:
8 """
9 class MyModule:
10     # Variables
11
12     # Methods
13     def __init__(self, input):
14         self.input = input
15         return
16
```

```

17 def run(self):
18     d = {}
19     d['pi'] = 3.14
20     return d

```

The essential elements of a module given below.

- In. 9. The module is defined in the class definition. By convention the module name has a capitalised first letter.
- In. 13-15. The initialisation method receives one argument: 'input', which is loaded into a class variables so that it can be accessed later. The variable 'input' is a DataContainer instance, which is the access point for input variables and all variables calculated by other modules. More on the use of DataContainers will follow in Section 3.5. Here it suffices to know that 'input' contains all the variables specified for this module in the input file and all variables returned by previously executed modules.
- In. 17. A module is required to contain a 'run' method that takes no arguments. This method is called by the modelling framework to execute the module.
- In. 18-20. A module should return a dictionary containing any number of entries (an empty dictionary is also allowed). The dictionary should contain the data that needs to be kept in memory after the module terminates. Other modules can access this variable using the key, e.g. in this case the variable 'pi' can be accessed from other modules returning a value 3.14.

Within the module it is allowed to define additional methods, import and call functions or classes or even call other programs or programming languages. It is strongly advised not to import or call other modules. With the freedom allowed in modules, a module can become very complex. There is nothing against complicated modules; iFlow is designed in such a way that collaborating programmers need not understand the implementation of a module. It is however highly recommended to add comments to the top of the module that state the aim of the module, its restrictions (e.g. a limited domain of application or assumptions), the input required and the output variables that will be calculated.

3.1.1 Submodules

Additional to modules, iFlow supports the optional feature of submodules. Submodules are independent, often similar, subtasks within a module. They are identified by a string name assigned to each submodule. As an example imagine a module that solves a linear equation $Au = b$ for u and for a set of right-hand sides b . One can define each of the right-hand sides as a submodule. Let us say that there are two right-hand sides and therefore two submodules which we will call 'effect1' and 'effect2'. The iFlow modelling framework supports a number of features concerning these submodules.

Firstly, to support submodules with highly similar tasks, iFlow facilitates a special data structure for submodules. In terms of the above example the two submodules 'effect1' and 'effect2' result in two solutions, say u_1 and u_2 . One can formally choose to save this in two ways. The straightforward way would be:

Saving results

■ Code sample 3.2 — Unrecommended way of saving results of submodules with similar tasks.

```

1 d={}
2 d['u_1'] = u_1
3 d['u_2'] = u_2

```

which saves two variables with the name 'u_1' and 'u_2'. This is quite clumsy and there is little motivation to the names 'u_1' and 'u_2'.

An alternative and preferable way of saving this data is the following:

■ **Code sample 3.3 — Recommended way of saving results of submodules with similar tasks.**

```

1  d={}
2  d['u'] = {}
3  d['u']['effect1'] = u_1
4  d['u']['effect2'] = u_2

```

This saves a single variable 'u' which is then separated into two subvariables: 'effect1' and 'effect2'. This way of saving data is generally cleaner, especially with a large number of submodules. Additionally, the access to this data (via a DataContainer, see Section 3.5) is more effective. The data can for example be accessed by calling the pair 'u', 'effect1', which returns the specific submodule, but can also be accessed by calling 'u' only. In this case the data of 'effect1' and 'effect2' are automatically summed and returned as a single variable.

Reduce calculations

Secondly, it can be specified in the input file which submodules need to be calculated. This information is passed to the variable 'submodules' available under the module input (i.e. the argument of the `__init__` method. If one for example only requires 'effect1' then 'submodules' is a list of one element, i.e. ('effect1'). This variable may be used to check which submodules need to be calculated and which do not. As such, it provides a tool to keep the number of calculations (and thus computation time) to a minimum.

Limiting input requirements

Finally, one can specify input requirements separately for each submodule. Imagine for example that 'effect1' requires no other modules to run first, while 'effect2' depends on the results of multiple other modules. When one is at some point only interested in the results of 'effect1', it is possible to request only 'effect1' in the input file. The modelling framework then automatically establishes that no other modules are required and will not run these modules. This can potentially reduce the complexity and computational time of a model significantly.

3.1.2 Module packages

Naming convention

Modules should be included in packages (i.e. a folder that contains an empty `__init__.py` file). It is recommended to create packages with a common theme or a common main author so that its easy to find modules. The names of packages should be unique and by convention should be lower case. Within a package, the names of modules should be unique, but module names can be reused for different packages.

Package location

Packages can be placed anywhere on the local machine or on a network disk. The package location can then be imported to iFlow via the input file. Three options apply:

- the iFlow packages folder. Packages placed in this folder are automatically imported to iFlow.
- your python source folder. Packages in the python source folder are automatically visible to python and therefore also available to iFlow.
- other locations. Packages on other locations can be imported to iFlow by including a special 'import (path)' statement in the input file. For example for a package 'mypackage' at location C:\MyStuff\Python\ the import statement is:

```
import C:\MyStuff\Python\mypackage
```

Other packages

Packages are not only limited to contain modules. A package that contains modules can also contain other python classes, functions or other files. It is also possible to create packages without modules, such as a toolbox package with general functions. The same rules for package locations apply to packages without modules. Packages can additionally contain any branching structure of sub-packages containing modules or any other file.

3.1.3 Registering modules

All modules should be registered in a special ‘registry.reg’ file. Each package has its own registry file, which should be placed in the root of the main package (i.e. not in sub-packages). The registry is iFlow’s reference to a module’s input requirements and expected return variables and is essential to the correct execution of the module.

The registry file is a simple ASCII file. A basic example of a registry file with two entries is given in Code sample 3.4.

Example
registry file

■ Code sample 3.4 — A registry file ‘registry.reg’.

```

1 # Registry file of package mypackage
2 # this package specialises in dummy packages used as an example
3
4 # Module MyModule #
5 module           MyModule
6 packagePath      folder1/
7 name             MyModule
8 input            foo bar
9 output           pi
10
11 # Module AdvancedModule #
12 module           AdvancedModule
13 input            grid pi
14 output           u
15 submodules       effect1 effect2
16 effect1         input
17                 output
18 effect2         input      var
19                 output

```

■

This code is explained below.

- In. 1, 2, 4, 11. Comment lines denoted by #
- In. 5,12. Start of a module entry, denoted by the keyword ‘module’. The value (‘MyModule’ or ‘AdvancedModule’) corresponds to the way this module is called.
- In. 6. Subfolder of the module within the package. ‘MyModule’ is located in a sub-package called ‘folder1’. ‘AdvancedModule’ is located in the root of the main package so that `packagePath` can be omitted.
- In. 8, 13, 16, 18. Input variables. These variables can be supplied in the input file, calculated by other modules or specified in the standard configuration file (see Section 3.3 for more information on sources of variables). The modelling framework uses this list to determine whether the input is complete and to determine the order in which to run modules. Note that input requirements can be defined for the whole module (all submodules) with separate additional requirements per submodule.
- In. 9, 14, 17, 19. Output variables provided by this module. This should list all the variables that are put in the dictionary returned by a module’s run method. Similar to input, output can be provided for the whole module (all submodules) with additional output per submodule. Note that the submodules ‘effect1’ and ‘effect2’ both output a variable ‘u’, which should then be saved according to the convention of Code sample 3.3.
- In. 15. List of submodules that only needs to be included if the module has submodules. ‘AdvancedModule’ has two submodules ‘effect1’ and ‘effect2’. The keywords ‘input’ and ‘output’ may be specified per submodule.

The keys ‘input’ and ‘output’ only specify the variable name and not its meaning, shape or data type. Please provide this information in the comment tag of the module or in a

manual corresponding to the module.

Advanced features

There are more options for registries, which are fully explained in Section 4.3. These options include tags for iterative modules, tags to force a module to run regardless of its output, tags for making the registry dynamically dependent on the input and a tag for indicating that a module writes output.

3.2 Input files

Location

The input file is the way to provide iFlow with information on which modules to load, which input parameters to use and which output to compute. The input file can be placed anywhere on the machine and can have any name and extension.

User interface

On starting iFlow, you will be prompted by a simple menu requesting the input file to start a simulation. The menu presents the five most-recently used input files and a user prompt.. There are three possible ways to answer this prompt:

- **press enter.** If no information, it will run the last used input file
- **enter a number between 1 and 5.** iFlow will run the corresponding input file from the list of most-recently used files.
- **file path incl. extension.** Enter an absolute file path or a path relative to the iFlow directory. The corresponding file is used.

Example input file

An input file is an ASCII file with information for every module. An example is given in Code sample 3.5.

■ Code sample 3.5 — Example input file.

```

1 # Input file
2 #
3 # Date:
4 # Authors:
5 import C:\MyStuff\Python\mypackage
6 import C:\MyStuff\Python\otherpackage
7
8 # MyModule
9 module      mypackage.MyModule
10 foo        always 10
11 bar        yes
12
13 # Load two modules at once
14 module      mypackage.AdvancedModule otherpackage.YetAnotherModule
15 submodules effect1
16 H          20
17 B          type   Polynomial
18           C      1000 0.1 0.01
19
20 # Output
21 module      general.Output
22 path        output/test
23 filename    my_output_file
24
25 requirements
26 variables   u v

```

Importing packages

This input file loads four modules: 'MyModule' and 'AdvancedModule' from the package 'mypackage', 'YetAnotherModule' from the package 'otherpackage' and a standard output module 'Output' from the package 'general'. On lines 5 and 6 we find import statements to import the packages 'mypackage' and 'otherpackage'. The package

'general' is not imported explicitly and should therefore be at one of the readily imported locations (see Section 3.1.2).

The input file consists of blocks of modules started by the keyword 'module'. For readability these blocks are separated by comment lines signalled by the tag #. The keyword module can be followed by one or more modules. Modules are loaded as (package-name).(modulename) (see e.g. ln. 8). The module block contains the variables that the module(s) require from input. For example in the first block 'MyModule' specifies two input variables 'foo' and 'bar' (ln. 8-10). The second block loads two modules 'AdvancedModule' and 'YetAnotherModule'. This block therefore contains all the variables that these two modules need; in this case two variables 'H' and 'B' (ln. 13-17).

Module blocks

The input file supports several shapes of input data. Firstly, one can give a single string or scalar (ln. 10, 14). Secondly, one can provide a row of values such as in ln. 9. This will be saved as a list, i.e. 'always', 10]. Finally, one can provide a multi-line block of data such as in ln. 15-16. This will be saved as a dictionary, i.e. ('type': 'Polynomial', 'C': [1000 0.1 0.01]). We will see later in Section 3.5 that data can be retrieved from this block by using the pair ('B', 'type') or ('B', 'C').

Shape of input

The output is written by a module. The standard output module is 'Output' in the package 'general'. We provide more details on output in Section 3.8 and here only discuss the way output is requested in the input file. Input variables needed by the standard Output module include 'path' and 'filename'. The variable 'path' specifies the output folder, while 'filename' (without extension) provides the preferred name of the output file. If the file name already exists in the output folder, the filename will be appended by the the first integer number > that creates a unique file name.

Specifying the output

The block of the Output module additionally contains the keyword 'requirements'. The information following this keyword is required by Output. The also allows the program to find the following information even if no output module is specified. The tag 'requirements' is followed by the mandatory tag 'variables' and the optional tag 'submodules'. The tag 'variables' indicates the variables to be written to output. The submodules to be written per module may be specified under 'submodules', with each variable and its submodules each on a separate indented line. The variables and submodules required on output will be used to determine which modules and submodules to run. The specification of output requirements is an important part of the input file. iFlow is designed to execute as few modules as possible to get the requested output. It will thus only execute those modules that calculate the output variables or that are needed somewhere in the process of calculating them. Conversely, iFlow will also provide a warning if you did not specify the proper modules to compute all the requested output.

Required variables

3.3 Variable names and sources

All quantities given on input, returned by modules or provided as configuration parameters are referred to as variables. Variables are identified using a name in a dictionary-like way; i.e. if one requests the variable name, say 'x', the system provides the variable value, say an array of floats (all details on this are provided in Section 3.5). This means that names of variables should be unique. If there are two modules providing some variable named 'u', then the second module will overwrite the entry of the first module. Similarly, if 'u' is also given on input, it will be overwritten by results of the modules.

As stated above, there are three ways of providing variables to the system:

1. returned by a module in a dictionary,
2. in the input file, or
3. in the standard configuration file.

This list also prescribes a hierarchy: if variables have the same name, then configuration variables are overwritten by input variables, which are overwritten by module output. This hierarchy can be used to one's advantage. Consider for example a variable, say `'omega'`, that has a generally fixed value and is therefore prescribed in the configuration file. If one then does a series of experiments with a different value of `'omega'`, this value can be given on input and the value from the configuration file is automatically overwritten. Be careful though that a variable is only ever overwritten by a variable of the same data type, i.e. float by float, array by array.

3.4 Dimensions and grids

One of the important features of almost any modelling problem are the dimensions (or axes) of the problem and grid on which its results are saved or calculated, in case of a numerical computation. In this section we will discuss how the dimensions and grids are defined in iFlow. We will also provide an example on how to design a module to make a grid.

3.4.1 Dimensions

The iFlow framework has some special functionalities concerning interpolation and conversion between analytical and numerical data. These functionalities require the system to have knowledge about the dimensions that are used and the names assigned to these dimensions. As an example let us imagine a modelling problem in two spatial dimensions and in a frequency domain. A sensible set of dimensions would then be `'dimensions' = ['x', 'y', 't']`. The program looks for `'dimensions'` in the dictionary variable `'grid'`. The variable structure thus reads `['grid']['dimensions'] = ['x', 'y', 't']`. The program then knows that we have a problem in three dimensions called 'x', 'y' and 't'.

Dimension
order
important

It is important to point out that **the order of the dimensions is important**. Numerical data sets should obey the order of the dimensions. This means that some function $f(t)$ translates to a numerical array of 3 dimensions. Its first 2 dimensions correspond to 'x' and 'y' and have length 1, because f is constant along the x and y axes. The third dimension corresponds to 't' and has a length greater than 1. So, for a run using 100 time steps, the numerical representation of f will have size $(1 \times 1 \times 100)$. iFlow allows that length-1 dimensions at the end are omitted. This means that a function $g(x)$ may be represented by a one-dimensional array. Similarly a function $h(y)$ may be represented by a two-dimensional array, where the first element is a length-1 dimension (i.e. no variation in x). Analytical functions (or numerical functions) allow for some flexibility and can work in a somewhat adjusted set of dimensions. More information on these shape rules is provided in Section 3.7

3.4.2 Grids

An iFlow program works principally with a single grid for calculations and a possibly different grid for storing its output. Additionally some support for multiple different calculation grids is offered in numerical functions (Section 4.1). Similar to any other variable, grids are identified by a string name. iFlow allows for the following options:

- `'grid'`. This is the standard calculation grid used throughout the program. It is used for numerical calculation of data and may be omitted in a model that uses only analytical functions.
- `'outputgrid'`. This is the grid that is used to save the result on. The output grid is required for every model.
- **other grids**. Other grids with custom names may be defined when e.g. reading data defined on a different grid, calculating a variable on a refined grid or using staggered grids. The use of other grids is supported through Numerical Functions,

which will be discussed in Section 4.1.

The program supports regular collocated non-equidistant grid axes. In the case of multi-dimensional grids, a curvilinear grid is created. In order to illustrate the restrictions and possibilities we take a two-dimensional example of axes (x, y) . The grid is constructed from two independent grid axes with the only exception that the boundary of x may depend on y and vice versa. This is illustrated in Figure 3.1. This must be in such a way that the corner points form a rectangle (figure a). If the corners do not form a rectangle (figure b)), a ghost corner should be constructed. The relevant boundary should then be extrapolated and it should be straight. The grid is fixed after its constructed and cannot be dynamically adjusted to calculated variables. This implies e.g. that σ -grids are not supported.

Supported
grid type

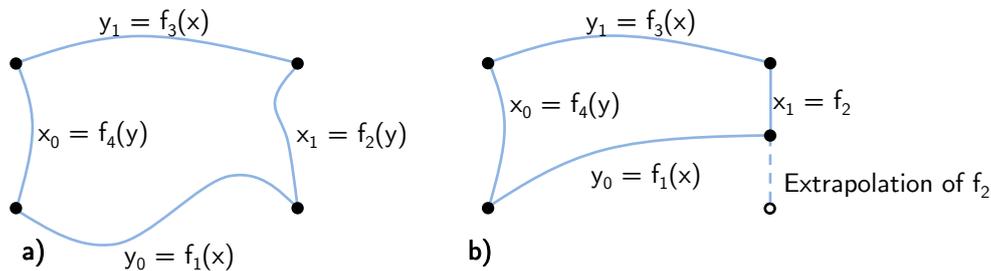


Figure 3.1: Illustration of two allowed domains. The boundary of x may depend on y and vice versa. This must be in such a way that the corner points form a rectangle (figure a). If the corners do not form a rectangle (figure b)), a ghost corner should be constructed. The relevant boundary should then be extrapolated and it should be straight.

The grid is saved as separate grid axes. Each axis is defined as a set of dimensionless points between 0 and 1 and a dimensional specification for the boundaries. There are several readily implemented axis types, which are identified by a string name. These are:

Supported
axis types

- 'equidistant'. Equidistant axis with grid points on the boundaries. It takes the maximum grid index as argument
- 'logarithmic'. Points are distributed as $(e^{\gamma X} - 1) / (e^{\gamma} - 1)$, where X is a set of equidistant points and γ is a steepness factor. It takes the maximum grid index and steepness factor as arguments
- 'list'. Define axis directly by prescribing the coordinates. It takes a list of either dimensional or dimensionless values as input
- 'integer'. Axis with integer steps, practical for discrete dimensions. This axis type is an exception as it is not saved as points between 0 and 1 and does not require boundary definitions.
- 'file'. Reads grid points from an ASCII file. The file path should be given as an argument. The file should contain a single column of grid points between 0 and 1.

Additional standard grid axes can be defined in the function

```
src.Tools.Grid.makeCollocatedGrid.
```

The axes (except 'integer' axes) are saved as dimensionless points between 0 and 1 accompanied by a boundary definition. So, in order to convert the grid axes to dimensional curvi-linear coordinates, one needs to combine the axes and boundary definitions. The full set of data stored in a grid definition is given in Table 3.1

Grid data
structure

The variable 'contraction' needs some explanation. The contraction denotes how the axis-boundaries depend on other variables. Here it says that the boundaries of the first dimension (x) depend on y , that the boundaries of the second dimension (y) depend on x and that the boundaries of the third dimension has no dependencies.

key	sub-key	sub-sub-key	example value	explanation	
'grid'	'gridtype'		'Regular'	only 'Regular' implemented	
	'dimensions'		['x', 'y', 't']	any list of string names of dimensions	
	'axis'	'x'		array(501)	grid points between 0 and 1.
		'y'		array(1, 201)	subkeys correspond to dimensions.
		't'		array(1,1,101)	the example is of a (501 × 201 × 101) grid.
	'low'	'x'		function(y)	boundary at grid points 0.
		'y'		function(x)	example is of grid in Figure 3.1b.
		't'		None	the example time axis is an integer axis.
	'high'	'x'		function(y)	boundary at grid grid points 1.
		'y'		function(x)	
		't'		None	
	'maxindex'	'x'		500	maximum index of each grid axis.
		'y'		200	note this is 1 smaller than the length.
't'			100		
'contraction'			[['y'], ['x'], []]	maximum index of each grid axis.	

Table 3.1: Grid data structure

3.4.3 Example module for defining dimensions and grids

iFlow contains a predefined function that makes it easy to define a grid. This function is named `src.Tools.Grid.makeRegularGrid`. Nevertheless, every model needs a special module to construct the grid, because the type of dimensions, axes and domain boundaries depend on the model. To assist building such a module Code sample 3.6 shows an example of a module for defining dimensions and grids. Code sample 3.7 provides the corresponding input file. The example corresponds results in the data structure of Table 3.1. Some important aspects of the code are explained below.

■ Code sample 3.6 — Example module for defining dimensions and grids.

```

1  """
2  GridDesign
3  Example module for defining the dimensions and grids
4
5  Date: 01-09-15
6  Authors: Y.M. Dijkstra
7  """
8  from src.Tools.Grid import makeRegularGrid
9
10
11 class GridDesign:
12     # Variables
13
14     # Methods
15     def __init__(self, input, submodulesToRun):
16         self.input = input
17         return
18
19     def run(self):
20         """Prepare a regular grid
21
22         Returns:
23             Dictionary containing:
24                 grid: dimensions (list) - list [x,y,t]

```

```

25         gridType (str) - 'Regular'
26         axis: x,y,t (ndarray) - grid axes between 0 and 1
27         high: x,y,t (any) - dimension-full limit belonging to point 1
28         low: x,y,t (any) - dimension-full limit belonging to point 0
29         maxIndex: x,y,t (int) - maximum index number
30     outgrid: idem
31     """
32     d = {}
33
34     dimensions = ['x', 'y', 't']
35     enclosures = [(self.input.v('lower_x'), self.input.v('upper_x')),
36                  (self.input.v('lower_y'), self.input.v('upper_y')),
37                  None]
38     contraction = [['y'], ['x'], []]
39
40     # Define 'grid'; the standard calculation grid
41     axisTypes = []
42     axisSize = []
43     variableNames = ['xgrid', 'ygrid', 'tgrid']
44     for var in variableNames:
45         axisTypes.append(self.input.v(var)[0])
46         axisSize.append(self.input.v(var)[1])
47     d['grid'] = makeRegularGrid(dimensions, axisTypes, axisSize, enclosures)
48
49     # Define 'outputgrid'; the grid for storing data
50     axisTypes = []
51     axisSize = []
52     variableNames = ['xoutputgrid', 'youtputgrid', 'toutputgrid']
53     for var in variableNames:
54         axisTypes.append(self.input.v(var)[0])
55         axisSize.append(self.input.v(var)[1])
56     d['outputgrid'] = makeRegularGrid(dimensions, axisTypes, axisSize, enclosures)
57
58     return d

```

Note in this example code

- In. 34. The dimensions are hard-coded here. This choice is made here because we want to fix the dimensions and the names of this model. Alternatively, it is possible to work with variable dimension names that are given in the input file.
- In. 35-37. The `enclosures` variable loads a list of tuples with lower and upper boundaries. These boundaries have been determined in a different module. Note that no boundaries are needed for dimension 't' that we fix as integer axis. The command `self.input.v(...)` retrieves the required boundary data. This command is explained in detail in Section 3.5.
- In. 41-46. Get the axis types and lengths from the input file.
- In. 47. Use `src.Tools.Grid.makeRegularGrid` to construct the grid and save this under the key 'grid'. This makes the standard calculation grid.
- In. 49-56. Do the same as in In 40-46, but now for 'outputgrid'. This makes the grid for saving data.

■ Code sample 3.7 — Part of input file for module GridDesign.

```

1  **GridDesign**
2  module      mypackage.GridDesign
3  xgrid       equidistant    500
4  ygrid       equidistant    200
5  tgrid       integer        100
6
7  xoutputgrid equidistant    50

```

```

8 youtputgrid  equidistant  50
9 toutputgrid  integer      100

```

■

3.5 Accessing data

The data access structure is one of the unique selling points of the iFlow modelling framework. iFlow provides a data wrapper class `DataContainer` that provides a number of options for accessing different data types in the same way. Among these data types are numerical arrays and analytical function prescriptions, so that its easy to switch between analytical and numerical calculation methods, without affecting the way data should be retrieved. This section will show the ways of accessing data via the `DataContainer` class.

Data types

We start by providing an overview of the data types and examples of their use in Table 3.2.

Data type	Use
Scalar/string	From single input/output value.
List	From multiple input/output values. Do not use lists for a variable that varies along a dimension (meaning axis), but use numpy ndarrays instead.
Numpy ndarray (in short: array)	The preferred data type for numerical data that varies along one or more dimensions. Arrays are never generated from input file data.
Function reference	For specifying analytical functions of any number of variables and for numerical functions (will be introduced in Section 4.1).
Dictionaries	Cannot be retrieved directly using a <code>DataContainer</code> . The <code>DataContainer</code> class is designed to retrieve values on the basis of one or more keys and other arguments. It is therefore in some sense an extension of the dictionary data type.

Table 3.2: Python data types and their use in iFlow.

3.5.1 The `.v()` method

The most important `DataContainer` method for retrieving data is `.v()` (where `v` is short for value). The `DataContainer` is based on the principal of dictionaries: the value is found by requesting a key. The `DataContainer .v()` method extends this principle by making it possible to look up keys with possible sub-keys, coordinates, array indices or other arguments. We will illustrate this by the example data structure of Table 3.3.

Command structure

Let us assume that the example of Table 3.3 is saved in a class variable `self.input`. The structure of a `.v()` call is then:

```
self.input.v(key, [(opt) subkey(s)], [(opt) coordinates/indices]).
```

key	sub-key	sub-sub-key	value	
'grid'	'gridtype'		'Regular'	
		'dimensions'	['x', 'y', 't']	
		'axis'	'x'	array(501)
			'y'	array(1, 201)
			't'	array(1,1,101)
	'low'		'x'	function(y)
			'y'	function(x)
			't'	None
	'high'		'x'	function(y)
			'y'	function(x)
			't'	None
	'maxindex'		'x'	500
		'y'	200	
		't'	100	
'foo'			['always', 10]	
'bar'			'yes'	
'H'			20	
'B'	'type'		'Polynomial'	
	'C'		[1000, 0.1, 0.01]	
'u'	'effect1'		3d ndarray (501 × 201 × 101)	
	'effect2'		3d ndarray (501 × 201 × 101)	
'v'	'effect1'		function reference (2d function)	
	'effect2'		2d ndarray (501 × 201)	

Table 3.3: Example data structure

First we will look closer at a `.v()` call for data that does not vary with the dimensions `x`, `y` or `t`. These data are generally either strings or lists and sometimes scalars. We give a number of examples:

■ Code sample 3.8

```

1 self.input.v('foo')
2   > ['always', 10]
3 self.input.v('foo', 0)      # use index to get first element from list 'foo'
4   > 'always'
5 self.input.v('B', 'C')     # use key and sub-key
6   > [1000, 0.1, 0.01]
7 self.input.v('B', 'C', 1)  # use key, sub-key and index to get 2nd element from ('B', 'C')
8   > 0.1
9 self.input.v('bar')
10  > 'yes'

```

Next we look at the response to 'illegal' calls. These are either calls to a non-existing key or calls that would return a dictionary. In the case of non-existing keys, the `DataContainer` will return `None` to signal nothing was found. In the case where a dictionary is found, the `DataContainer` will not return that dictionary. Instead it will return `True` to signal something was found, but may not be returned. Some examples are given below

■ Code sample 3.9

```

1 self.input.v('Q')
2   > None
3 self.input.v('effect1')
4   > None      # 'effect1' is not a main key (only a sub-key)
5 self.input.v('B')
6   > True
7 self.input.v('grid')

```

8 > True

Finally we look at the calls to dimension-varying data. This data will come in the form of arrays, function references or scalars (i.e. scalars indicate that a quantity is constant in all dimensions). The data can be requested in three ways:

- **coordinates**. Request data on one or more (x, y, t)-coordinates. Numerical data will automatically be interpolated to the requested coordinates. Functions will be evaluated on the requested coordinates. Note that the (x, y, t)-coordinates should be dimensionless coordinates between 0 and 1 (with exception of the integer axis t, see Section 3.4).
- **grid indices**. Request data using integer grid indices. Numerical data will simply be returned on the indices. Functions will be evaluated at the (x, y, t)-coordinate corresponding to the grid indices.
- **no specification**. If no coordinates or indices are specified, the original format will be returned. Numerical data will be returned as full array and functions are returned as function references. This functionality should mainly be used for testing whether a variable exists. It can also be used when one wants data on the full grid and is 100% sure that the data is an array.

The data returned will always try to shape itself according to the requested shape. This means that one can request the 2D variable 'v' on the full 3D grid. The variable will then copy its values for all t-values. If a variable cannot shape itself according to the request in some dimension, it will try to return its original form in that dimension.

We illustrate this by giving a wide range of examples. We give these examples in the form of Code sample 3.10.

■ **Code sample 3.10 — Requesting dimension-varying data.**

```

1 #####
2 # using no specification
3 #####
4 x = self.input.v('grid', 'axis', 'x')
5 y = self.input.v('grid', 'axis', 'y')
6 t = self.input.v('grid', 'axis', 't')
7     > x = array(501)           # returns full numerical array.
8     > y = array(1, 201)       # This is ok; axis is always an array.
9     > t = array(1, 1, 101)
10 xmax = self.input.v('grid', 'maxIndex', 'x')
11 ymax = self.input.v('grid', 'maxIndex', 'y')
12 tmax = self.input.v('grid', 'maxIndex', 't')
13     > xmax = 500
14     > ymax = 200
15     > tmax = 100
16
17 u_1 = self.input.v('u', 'effect1')
18 v_1 = self.input.v('v', 'effect1')
19 v_2 = self.input.v('v', 'effect2')
20     > u_1 = array(501,201,101) # returns full array.
21     > v_1 = function           # returns function reference.
22     > v_2 = array(501,201)     # v_2 is only a 2d array.
23                               # NB results are inconsistent; so only recommended
24                               # for checking that u and v exist.
25
26 u = self.input.v('u')
27     > u = array(501,201,101)   # automatically adds 'effect1' and 'effect2'
28                               # then returns full array.
29 v = self.input.v('v')
30     > v = True                 # Tries to add 'effect1' and 'effect2',

```

```

31             # but cannot add array + function.
32             # Therefore finds a dictionary.
33
34 #####
35 # using coordinates
36 #####
37 # Coordinates are indicated by using x=., y=., t=..
38 u_1 = self.input.v('u', 'effect1', x=0.103, y=0.513, t=10) # x,y between 0 and 1,
39 v_1 = self.input.v('v', 'effect1', x=0.103, y=0.513, t=10) # t is an integer axis.
40 > u_1 = 0.824 # value of u at this point; data is interpolated.
41 > v_1 = 0.532 # value of v at this point; function is evaluated.
42             # note that 'v' has no t-axis, but accepts t=10 in the call.
43
44 u = self.input.v('u', x=0.1, y=0.2, t=10)
45 v = self.input.v('v', x=0.1, y=0.2, t=10)
46 > u = 1.213 # adds 'effect1' and 'effect2'.
47 > v = 0.759 # now succeeds at adding 'effect1' and 'effect2'.
48
49 u = self.input.v('u', x=np.arange(0,1,0.1), y=np.arange(0,0.2,0.1), t=10)
50 u = self.input.v('u', x=np.arange(0,1,0.1), y=np.arange(0,0.2,0.1), t=[10])
51 > u = array(10, 2) # 'effect1'+ 'effect2' on requested points; interpolated.
52 > u = array(10, 2, 1) # uses standard numpy rules:
53 # request t as scalar -> eliminate dimension,
54 # request t as list/array -> keep dimension.
55
56 v = self.input.v('v', x=np.arange(0,1,0.1), y=np.arange(0,0.2,0.1), t=np.arange(0,10))
57 > v = array(10, 2, 10) # note that 'v' has no t-axis.
58 # It will try to fit to the size of the request
59 # by copying its values for all t.
60
61 H = self.input.v('H', x=np.arange(0,1,0.1), y=np.arange(0,0.2,0.1), t=np.arange(0,10))
62 > H = array(10, 2, 10) # also the scalar H will be copied to fit
63 # the size of the request.
64
65 v = self.input.v('v', x=x, y=y, t=t)
66 > v = array(501, 201, 101) # evaluate v on full grid using
67 # x, y, t in ln. 4-6.
68 # Adds 'effect1' and 'effect2'
69
70 #####
71 # using indices
72 #####
73 # Indices are indicated by NOT using x=., y=., z=..
74 # The same rules apply as when using coordinates
75 u_1 = self.input.v('u', 'effect1', 3, 198, 53) # indices on the grid
76 v_1 = self.input.v('v', 'effect1', 3, 198, 53)
77 > u_1 = 0.654 # value of u at this point.
78 > v_1 = 0.465 # value of v at this point; function is evaluated
79
80 u = self.input.v('u', x=range(0, 20), y=range(0, 42), t=10)
81 u = self.input.v('u', x=range(0, 20), y=range(0, 42), t=[10])
82 > u = array(20, 42) # 'effect1'+ 'effect2' on requested points
83 > u = array(20, 42, 1)
84
85 v = self.input.v('v', x=range(0, xmax+1), y=range(0, ymax+1), t=range(0, tmax+1))
86 > v = array(501, 201, 101) # evaluate v on full grid using
87 # xmax, ymax, tmax in ln. 10-12.
88 # Adds 'effect1' and 'effect2'

```

3.5.2 The `.n()` and `.d()` methods

The methods `.n()` and `.d()` compute the negative and derivative respectively. These commands are closely related to `.v()` and inherit all of its functionality. The `.n()` command will try to return -1 times the value, if the return value is a scalar, array or function. If the return data is a list, string, True or None, it will not alter this.

The `.d()` command computes the derivative of a value if the value is a scalar, function or array. If the derivative of an analytical function has been implemented, `.d()` will use this¹. Else, if the data is numerical, `.d()` will compute the numerical derivative.

It is **recommended to use indices instead of coordinates** with `.d()` if possible. This is because the use of coordinates is inefficient when requesting data from an array on few points. The method will then compute the derivative of the numerical array on the whole grid and then interpolate.

A call to `.d()` requires the addition of the dimension to take the derivative in. The call looks like

```
self.input.d(key, [(opt) subkey(s)], [(opt) coordinates/indices],
dim=name(s) of dimension(s)).
```

The added argument `dim` takes the scalar name(s) of the dimension(s) with respect to which the derivative is required. In case of higher-order derivatives `dim` can take a string of names, such as `'xx'` to denote the second derivative with respect to `x`, or `'xy'` to denote the mixed derivative.

3.5.3 Other methods

slice	The <code>slice(key, [(opt) sub-key(s)])</code> method returns a new <code>DataContainer</code> instance containing only the variable requested with the arguments 'key' and optional 'sub-key(s)'. This is for example useful when one needs to provide a function with the grid (in a <code>DataContainer</code>) as an argument. The call <code>self.input.v('grid')</code> then returns a new <code>DataContainer</code> containing only the grid and all its sub-keys.
addData	The <code>addData('key', 'value')</code> statement adds 'value' to the <code>DataContainer</code> under the name 'key'. This command does not allow for specifying sub-keys. This is however still possible making 'value' a dictionary. For example the statement <code>addData('B', dict('C2': [0, 1]))</code> creates a key 'B', sub-key 'C2' and value [0, 1].
merge	The <code>merge(dc)</code> statement merges <code>DataContainer</code> instance <code>dc</code> with this <code>DataContainer</code> instance. If <code>dc</code> contains duplicate keys, then it will overwrite the keys in <code>self.input</code> . This will be done on the lowest sub-key level. For example if <code>self.input</code> contains ('u', 'effect1') and <code>dc</code> contains ('u', 'effect2'), then the resulting container will contain both.
copy	The <code>copy()</code> method returns a shallow copy of the <code>DataContainer</code> . This means that the data inside the <code>DataContainer</code> is not copied, but that it is put into a new <code>DataContainer</code> instance.
getAllKeys	The command <code>getAllKeys()</code> returns a list with all keys. Key-subkey pairs are returned as tuples in this list.
getKeysOf	The command <code>getKeysOf(key, [(opt) subkeys])</code> returns a list with all keys under the argument key and possible sub-keys. This method only returns one level of keys and not its sub-keys.

¹.`d()` will give an error when the derivative of an analytical function is not implemented. `iFlow` does not automatically switch to numerical derivatives.

3.6 Specifying analytical functions

iFlow contains a set of functionalities that make it easy to define analytical functions and make them accessible from a `DataContainer`. An analytical function should be defined as a class that extends (i.e. is a sub-class of) `src.FunctionTemplates.FunctionBase`. This `FunctionBase` superclass implements all the functionalities to make a function accessible from a `DataContainer`. Function as sub-class

A function class should contain at least two methods and supports one standard optional method: Methods

- `__init__(self, dimNames, parameters)`.
- `value(self, [(opt) dimensions], kwargs)`.
- optional `derivative(self, [(opt) dimensions], kwargs)`.

Additionally one is free to add any other method to a function class and call these methods from either one of the above methods. The arguments have the following meaning:

- `dimNames` fixes the names of the dimensions of this function when the function is instantiated. It should be given as a single string (1 dimension) or a list of strings (multiple dimensions). As a consequence, e.g. a function of 1 variable can be implemented using a variable `x`, but this does not fix this function to actually vary with the dimension named `'x'`. The function class can be instantiated using e.g. `dimNames='y'`. At that moment, the function is made a function of `'y'`. The `dimNames` have to correspond to names from the defined dimensions. If `dimNames` contains multiple elements, the order should be the same as defined in the dimensions, but omitting unnecessary dimensions.
- `parameters` can be any parameter or set of parameters (in list, tuple, dictionary or `DataContainer`). The `__init__` method should load these to public class variables.
- `dimensions` is a single argument for a function of 1 variable and are multiple arguments for multiple variables. It should be given as either a scalar, list or array. These are the points to evaluate the function on. The arguments can be either scalar or array **make sure that the function can handle both scalar and array arguments**.
- `kwargs` should always be accepted by the methods `value` and `derivative`, even if not used. The method `derivative` has one standard entry in `kwargs`: `'dim'`. This gives the dimensions of the derivative, e.g. `'xx'` for the second derivative with respect to `x`.

An example of a function class `Polynomial` in 1 variable is given in Code sample 3.11. Example function

■ Code sample 3.11 — Example of a polynomial function in one dimension.

```

1 """
2 Polynomial function for 1 variable between 0 and 1
3 Implementation of FunctionBase.
4
5 Requires parameters  'C': list/array of coefficients of the polynomial
6                      ranging from the higher to the lowest order term
7                      The length of the list sets the order of the polynomial
8                      'L': length of system
9
10 Date: 23-07-15
11 Authors: Y.M. Dijkstra, R.L. Brouwer
12 """
13
14 import numpy as np
15 from src.FunctionTemplates.FunctionBase import FunctionBase
16
17
18 class Polynomial(FunctionBase):
19     #Variables
20
21     #Methods

```

```

22 def __init__(self, dimNames, parameters):
23     FunctionBase.__init__(self, dimNames) # instantiate superclass & fix dimNames.
24     self.L = parameters.v('L')           # this functions uses parameters L and C,
25     self.C = np.array(parameters.v('C')) # which should be given in a
26                                         # DataContainer 'parameters'.
27
28     # FunctionBase provides a method to check if indeed C and L have been set:
29     FunctionBase.checkVariables(self, ('C', self.C), ('L', self.L))
30     return
31
32 def value(self, x, **kwargs):
33     """
34     Parameters:
35         x - value between 0 and 1
36     """
37     x = x*self.L
38     return np.polyval(self.C, x)
39
40 def derivative(self, x, **kwargs):
41     """
42     Parameters:
43         x - value between 0 and 1
44     """
45     x = x*self.L
46     Cx = np.polyder(self.C)
47     if kwargs['dim'] == 'x':
48         return np.polyval(Cx, x)
49     elif kwargs['dim'] == 'xx':
50         Cxx = np.polyder(Cx)
51         return np.polyval(Cxx, x)
52     else:
53         FunctionBase.derivative(self) # makes sure to return a useful error message
54     return

```

■

Assigning a
function

Code sample 3.12 shows how a function class can be assigned to a variable and then used. For the example imagine that the polynomial function defined above is needed for a height profile variable named 'height'.

■ **Code sample 3.12 — Assigning and using a function.**

```

1 from functions.Polynomial import Polynomial
2
3 # First instantiate the analytical function class using a temporary variable poly.
4 # It is made a function of 'y'. self.input is a DataContainer containing L and C.
5 # Then assign it to height using .function (no brackets)
6 poly = Polynomial('y', self.input)
7 height = poly.function
8
9 # We can use the function by first putting it in a DataContainer
10 # and then calling its data
11 self.input.addData('height', height)
12 h_1 = self.input.v('height', y=0.5)
13 h_y = self.input.v('height', y=np.arange(0,1,0.1))
14 > h_1 = 3.53
15 > h_y = array(10)

```

■

3.7 Shape of array data

Many of the variables in a model will be provided in the form of arrays. Arrays represent data that varies in one or more of the model dimensions and typically result from numerical computations or evaluating analytical functions. The allowed shape of arrays is restricted. We will discuss these shape restrictions in this section.

Firstly, the shape of the array should respect the order of the dimensions. In our example where `dimensions = ['x', 'y', 't']`, a variation of an array in x-direction should always be represented in the first argument of the array, a variation in y-direction in the second argument and a variation in t-direction in the third argument. Variations of the array variable over other dimensions are allowed in additional arguments, but this should be used only within a single module and not as output of a module. Note also that additional dimensions in arrays can only be read by using indices, not using coordinates. respect dimensions

It is important to realise that (analytical) functions operate on a potentially different set of dimensions than the above `dimensions` variable. As such, functions provide a different environment to work in. The dimensions of a function are set on instantiation. These dimensions should still obey the order of the `dimensions` variable, but dimensions may be left out. New dimensions may be added at the end, but again this functionality should only be used within a single module. Such additional dimensions in functions can only now be read by using coordinates, not indices. dimensions in functions

Secondly, numerically computed data should respect the grid. This means that the number of elements in each dimension of an array should be one of the following: respect grid

- the same as the number of grid points in each dimension, or
- 1 if the the array variable does not vary in this dimension, or
- nothing if the array variable does not vary in this dimension and this dimension is at the end of the array (i.e. length-1 dimensions at the end may be omitted).

We provide a few examples to illustrate the shape rules

■ Code sample 3.13 — Shape rules for arrays.

```

1 # Let dimensions = ['x', 'y', 't'] and the grid be (501, 201, 101) in size
2 # Let furthermore u(x, y, t), v(x, y), w(x, t), q(x, lambda)
3 u.shape()
4     > array(501, 201, 101)
5 v.shape()
6     > array(501, 201)
7 w.shape()
8     > array(501, 1, 101)
9 q.shape()
10    > array(501, 1, 1, 50)    # assuming dimension lambda has length 50
11
12 # Let f be a function of (x, t): f(x,t) = x*t
13 Class MyFunction(FunctionBase):
14     def __init__(dimNames)
15         FunctionBase.__init__(self, dimNames)
16         return
17
18     def variable(a, b):      # note: names of variables can be anything here
19         a = np.asarray(a).reshape(len(a), 1)    # need to adjust a, b to the right shape
20         b = np.asarray(b).reshape(1, len(b))
21         f = a*b                # f has the right shape: (len(a), len(b))
22         return f
23
24 # This function is instantiated and called as:
25 myfun = MyFunction(['x', 't'])    # instantiate
26 self.input.addData('f', myfun.function)    # add to DataContainer

```

```

27 f = self.input.v('f', range(0,10), range(0,20))
28 > f = array(10, 1, 20)      # note that the shape of f is different than in the
29                             # MyFunction class.
30                             # The DataContainer transforms the shape of the
31                             # function environment ['x', 't'] to the normal
32                             # dimension environment ['x', 'y', 't']

```

■

3.8 Output storage

iFlow allows you to store calculated variables using an output module. The standard output module `general.Output` takes two mandatory input variables. Firstly, the path (relative or absolute) to the output folder. The path will be created if it does not yet exist.

Secondly, the file name without extension. The file name allows for dynamic naming, which allows the file name to contain values of one or more variables through the command `varname', options@'`. For example, let us consider variables `'L'` equal to 10000 and `'Amp'` equal to `[0, 1.5, 0, 0]`. Then the dynamic file name `L'Amp', 1my_output_file_L@'_A@'` will result in `my_output_file_L10000_A1.5`. The code between the accolades uses calls similar to those used in the `DataContainer v()` method, i.e. as first key it takes the variable name and options can include subkeys, indices or coordinates. The variable used in the dynamic name may be given in the configuration file, input file or the result of module calculations. If the file name already exists, iFlow appends the requested file name with the lowest possible integer, so that files are never overwritten.

Optional variables

The standard output module also allows for the optional arguments `iteratesWith`, `saveAnalytical` and `dontConvert`. `IteratesWith` is followed by the name of an iterative module (see also Section 4.2) and results in output being written in every loop of the iterative module. This is e.g. useful for a batch computation. `saveAnalytical` is followed by variable names. The program will try to save these variables analytically, which means that a reference to the analytical function is written including all the class variables of this function. Finally, `dontConvert` guarantees that the variable is saved without converting data to the output grid.

■ Code sample 3.14 — Output module in the input file.

```

1 **Output**
2 module      main.Output
3 path        output/test
4 filename    my_output_file_L@{'L'}_A@{'Amp', 1}
5 iteratesWith general.Sensitivity      # optional
6 saveAnalytical v                      # optional
7 dontConvert      u                    # optional
8
9 requirements    u v

```

■

Output grid

The output module requires a grid with the name `'outputgrid'`. The program will try to convert all variables to the output grid, interpolating data when necessary. A variable will only not be converted to the output grid if it is not grid-conform or if it is listed under `dontConvert`. In general, (analytical) functions will be evaluated on the output grid before saving.

(Numerical) functions

Derivatives of analytical functions are not saved. Derived quantities in numerical functions on the other hand are saved on the output grid.

Pickle

The output is written to file using the `Pickle` module in Python. The resulting files are binary

files that can be loaded back to python using the unpickle functionality of the Pickle module. During output writing, the output is stored in a dictionary format with a slightly different structure than in the DataContainer. The NiFTy function `pickleload` may therefore be used to load stored data back into iFlow. Alternatively the readily available modules `LoadSingle` and `LoadMultiple` are available to load one or multiple files back into iFlow respectively.

4. Advanced features

4.1 Numerical functions

Numerical functions are a special feature of iFlow that combine the functionality of analytical functions with numerical data. Numerical functions are especially useful in two cases:

1. The numerical data comes on a different grid than the standard grid, e.g. when loading data from measurements or another model, or when computing some quantity on a refined grid.
2. You would like to save the numerical data with its (analytically calculated) derivative or second derivative.

Numerical functions take the interface of analytical functions (Section 3.6) and thus look like a function on the outside. Internally, they contain a `value` method and optionally `derivative` and `secondDerivative` methods. These methods do however not return a function prescription, but call numerical data that is defined on some grid. The data will be interpolated if necessary using the `DataContainer`'s call-by-coordinate functionality.

4.1.1 Using loading data into a numerical function: `NumericalFunctionWrapper`

A simple interface exists to transfer already computed or read data to a numerical function in one to three lines of code. This interface does not require you to construct a new numerical function class or method, but simply instantiates the `NumericalFunctionWrapper` class. We illustrate this in Code sample 4.1.

■ **Code sample 4.1 — Loading data to a numerical function.**

```
1 # After calculating a variable u and its derivatives u_x and u_xx we load this to a numerical function
2 from src.FunctionTemplates.NumericalFunctionWrapper import NumericalFunctionWrapper
3 nf = NumericalFunctionWrapper(u, self.input.slice('mygrid'), 'mygrid')
4 nf.addDerivative(u_x, 'x')
5 nf.addSecondDerivative(u_xx, 'x')
6
7 d = {}
8 d['u'] = nf.function
```

The NumericalFunctionWrapper is instantiated in line 3. This instantiation requires three variables:

- the `variable`. A numerical function can only contain one variable, possibly with its derivatives and second derivatives.
- the `grid`. The grid should be loaded to the numerical function as a DataContainer. A grid can be isolated from a bigger DataContainer instance using the `.slice('varname')` method of the DataContainer. This method creates a new DataContainer instance containing only the variable with name 'varname'. In the above example we use the grid with name 'grid', which is the standard grid. However, any grid can be defined and transferred to the numerical function (more on defining a grid in Section 3.4).
- the `grid name`. The grid name is the name under which the grid (of the second argument) is known in the DataContainer.

The numerical function automatically determines its dimensions from the variable 'dimensions' in its grid and the size of the numerical data, i.e. if 'dimensions': [x,y,t] and the numerical data has one dimension, then the numerical function is a function of 'x'.

Derivatives can be added using the `addDerivative` and `addSecondDerivative` commands (line 4 and 5), which take the variable and the dimension of differentiation as arguments. At the moment, the second derivative can only handle a double derivative along the same dimension.

4.1.2 Writing your own numerical function: NumericalFunctionBase

If you require more functionality than just loading data to a numerical function, it is possible to define your own numerical function that, for example, makes computations or reads data from a file. A custom numerical function can be created by defining a class that extends the NumericalFunctionBase superclass. Code sample 4.11 shows an excerpt of an example of a custom numerical function that loads data from a file. We explain all the functionalities below.

■ Code sample 4.2

```

1 """
2 Exert from a numerical function to read data from a file
3
4 Date: 10-09-15
5 Authors: Y.M. Dijkstra
6 """
7 from src.FunctionTemplates.NumericalFunctionBase import NumericalFunctionBase
8
9
10 class ReadTxt(NumericalFunctionBase):
11     #Variables
12
13     #Methods
14     def __init__(self, dimNames, data):
15         NumericalFunctionBase.__init__(self, dimNames)
16         self.__file = data.v('file')
17         self.__varname = data.v('varname')
18
19         # check the input.
20         NumericalFunctionBase.checkVariables(self, ('file', self.__file),
21                                             ('varname', self.__varname))
22
23         return
24
25     def __readtxt(self):

```

```

26     # ...
27     # ...all kinds of commands to read a from a file:
28     #     - a variable 'u'
29     #     - a DataContainer 'grid' containing everything one needs for a grid
30     # ...
31     self.addValue(u)
32     self.addGrid(grid)
33     return

```

init The `init` method of a numerical function class is the same as that of an analytical function (see also Section 3.6). The method takes two arguments. The first, `dimNames`, is a string dimension name or a list of string dimension names which fix the dimension names of the function at instantiation. The second, `data`, can be any (set of) parameter(s) in any format, but is preferably a `DataContainer`. The `init` method first needs to call the `init` of the super class (ln. 15). The rest of the method can be used in any way, but it is common to load the parameters to class variables (ln 16-17) and check whether they are properly set (ln 20).

Required commands

The numerical function should contain the following additional calls:

- `self.addValue(var)`. Add the return variable `var` to the numerical function.
- `self.addGrid(grid)`. Add a `DataContainer` `grid` to the numerical function. The variable and grid dimensions and size should correspond (see Section 3.7).
- (optional) `self.addDerivative(var, dim)`. Add derivative `var` along dimension `dim` to the numerical function.
- (optional) `self.addSecondDerivative(var, dim)`. Add second derivative `var` along dimension `dim` to the numerical function. At the moment the numerical function only supports differentiation twice along the same dimension.

These calls can be made from any method in the numerical function class (note this is different to analytical functions which require a `value()` method). The value or derivatives do not need to be returned by any method like in analytical function. The above commands will add your data to the numerical function and will make it available to access using a `DataContainer`.

Instantiating The instantiation of a custom numerical function class is exactly the same as an analytical function, see Section 3.6.

4.2 Iterative modules

methods

Iterative modules are modules that instruct the iFlow core to start an iteration loop over this and possibly other modules. Iterative modules have to satisfy the same criteria as normal modules, i.e. they contain an `__init__(input)` and `run()` method and the `run` method returns a dictionary, and additionally have to contain a method `run_init()` and `stopping_criterion(iteration)`. The method `run_init()` is called instead of the `run` method in the first iteration of a loop. Similar to the `run` method it needs to return a dictionary. This way, the module can perform a different set of tasks during the first iteration than during consecutive iterations, when the `run` method is called. The `stopping_criterion` method is called by the iFlow core after each call of `run_init` or `run`. As argument it takes the current iteration number. The `stopping_criterion` should return a boolean which is set to `True` to instruct the core that no more iterations are required and `False` to instruct the core to continue the loop.

registry

An iterative method further requires one extra line in its registry

```
iterative True.
```

If this line is missing, the core will treat the module like an ordinary module. The registry of the iterative module (and its submodules) may additionally contain the keyword `inputInit`.

This, like the keyword `input`, is followed by the variable names of the variables required as input to this model, but then specifically for the initial run of the module.

The place of an iterative module in the call stack is determined by the input requirements for the initial run. In other words, the module is placed in the call stack behind all the modules needed to satisfy the initial input requirements. If there are multiple modules that can be placed in the call stack at the same time (i.e. have the same input requirements), then iterative modules are placed after non-iterative methods. After an iterative module is placed in the call stack, the iFlow core determines the modules that need to be included in the iteration loop. The iteration loop contains all modules that come after the iterative module in the call stack and that are needed to fulfil the input requirements of the iterative module. That is, those modules that produce output that is in the iterative method's input requirements, but not in its initial input requirements.

call stack

iFlow can work with multiple iterative modules in one simulation, whether the iteration loops are consecutive or nested. In some cases, the combination of two iterative modules produce nested loops (i.e. a loop in a loop) where the requirements allow either loop to be the nested one. It may be obvious to a user that one of the two loops requires fewer iterations or is less complex and it is more efficient take it as the nested loop. The iFlow core may however not always be able to determine the most efficient loop, as it has no a-priori information on the size or complexity of the loops. Which loop is taken as the nested loop can then be manually adjusted in the input file. iFlow will try to use the iterative method that comes first in the input file as the initiator of the outer loop. Iterative methods coming later in the input file will initiate the nested loops.

manual adjustments

4.3 Dynamic options for registries

In some cases the input requirements or output of a module depend on the value of a parameter. As an example imagine a module called `TrickOrTreat` that takes an input variable `choice` with two possible values: 'trick' or 'treat'. In case it is set to 'trick', the module requires no input and outputs a variable `trick`. In case `choice` is set to 'treat', the module requires the input variable `candy` and outputs a variable `treat`. The registry files allow for a number of options to enable this dynamic behaviour. As an example, Code sample 4.3 shows two such options for the module `TrickOrTreat`.

■ Code sample 4.3 — Example of dynamic registry functions.

```
1 Registry entry for module TrickOrTreat
2
3 module    TrickOrTreat
4 input     choice if{@choice=='treat', candy}
5 output    @choice
```

Firstly, the symbol `@` in `@choice` in the code sample indicates that this entry needs to be replaced by the value of `choice`. The output of this module should thus be a variable with a name equal to the value of `choice`. Alternatively one can use the notation `@{...}`. This may be used in compound statements, where something needs to come before or after the dynamic statement. For example `no@{choice}` would yield 'notrick' or 'notreat' depending on the value of `choice`. Also one can indicate elements from a vector with this notation as `@{myvector, 1}`, which takes the element at index 1 of the vector with name 'myvector'. The notation `@{...}` automatically converts its result to a string. Therefore, even if `@{myvector, 1}` returns a number, say 2, then iFlow will interpret this as the string '2'. The original data types are conserved when `@` (i.e. without curly brackets) is used. In any case, the variables used after `@` should be provided as input to this module in the input file.

symbol @

The last statement holds for all but a final option involving `@`. One can also use the construct `@output.requirements`. This returns the list of variables behind the keyword 'requirements' in

the input file. Recall this keyword is used for the output requirements of a simulation and lists the minimum set of variables that should be computed. This can be useful e.g. for iterative modules that should set-up a loop over all the following modules. The `input` in the registry should then be equal to all variables that are computed in a simulation.

if-statement The second dynamic registry option is the construct `if{condition, consequence}` as in Code sample 4.3. In this example, if the condition is met, i.e. `@choice=='treat'`, the variable name `candy` is added to the list of input requirements. Nothing happens if the condition is not met. As the example illustrates, the dynamic notation with `@` may be used inside the if-statement.

symbol + Finally, variable names involving counters can be defined using a short-hand notation (not shown in the example). Variables with counters are e.g. `u0`, `u1`, `u2` etc. A list of these variables can be added to the input requirements or output of a module using the notation `variable+{start, stop, increment}`, e.g. `u+{0, 6}`, where the default increment is set to 1. This returns a list with elements 'u0' to 'u5'. This notation can also be made dynamic e.g. `u+{0, @{maxorder}, @{increment}}`, where the values of two input variables 'maxorder' and 'increment' are used to determine the list of input variables. Note that we use `@{maxorder}` and `@{increment}`, instead of `@maxorder` and `@increment` (i.e. we use curly brackets). This is required, because the values of 'maxorder' and 'increment' are needed as strings, so that they can be combined with the rest of the statement.

4.4 The NiFTy toolbox

The NiFTy (Numerical iFlow Tools) toolbox contains a diverse set of functions for operations on arrays. The functions in this toolbox may be used in modules and may be imported by importing a specific function from NiFTy or by importing the while package as `import nifty as ny`. The following sections present a selection of useful functions in the NiFTy toolbox.

4.4.1 Numerical differentiation

NiFTy implements numerical differentiation routines that are especially suited to computations within iFlow's data structure. The syntax is as follows

■ **Code sample 4.4 — derivative.**

```

1 derivative(u, dim, grid, (optional) indices, DERMETHOD=None)
2 '''
3 Takes the derivative of u along dimension dim
4 Parameters:   u: array
5               Array to take derivative of.
6               dim: string or int
7               String name of the dimension with respect to with the derivative
8               needs to be taken. The name should match the name definition
9               in the grid dimensions variable. Alternatively it can be the integer
10              position of this dimension is the grid dimensions list.
11              grid: DataContainer instance
12                  DataContainer containing a full grid variable (with all subvariables).
13                  Typically obtained by slicing the modules' main data container,
14                  e.g. self.input.slice('grid'). This splits off the part of the data
15                  container containing 'grid'. The grid dimensions and shape of u should
16                  match. Note: the grid may contain additional dimensions that u does
17                  not have and u may have additional dimensions that the grid does
18                  not have. In the latter case derivatives cannot be computed with
19                  respect to the additional dimensions.
20              indices: (optional) arrays or numbers
21                  One or multiple arguments stating the grid-indices along which the
22                  output should be returned (as in DataContainer). If not set, returns
23                  output along the entire grid
24              DERMETHOD: (optional) string
25                  Numerical derivation scheme. Currently allows for 'CENTRAL' and
26                  'CENTRAL2'. 'CENTRAL' is a standard central method that degrades
27                  to a first-order upwind method at the boundary. 'CENTRAL2' is
28                  identical in the interior, but is replaced by a second-order BDF scheme
29                  at the boundary. If None, the derivation method is taken from the
30                  iFlow standard configuration file (keyword DERMETHOD).
31 '''
```

Similarly, NiFTy contains a method for computing the second derivative along one axis

■ **Code sample 4.5 — secondDerivative.**

```

1 secondDerivative(u, dim, grid, (optional) indices)
2 '''
3 Takes the second derivative of u along dimension dim. Does not compute
4 cross-derivatives over multiple dimensions. See function derivative for
5 explanation of the input
6 '''

```

The second derivative is computed using a central method. At the boundaries it is assumed that the second derivative is identical to the second derivative one cell from the boundary.

The NiFTy implementations allow for two specific advantages over other methods

1. differentiation and integration over non-uniform grids, and
2. automatic adjustment of the derivative along a grid that is mildly sloping relative to the axis of derivation.

The latter is explained below.

iFlow works using dimensionless grid axis between 0 and 1. An axis is made dimensional using grid enclosures that may be a function of any other dimension, creating a curvilinear grid. A property of curvi-linear grids is that neighbouring grid points are not necessarily on the same Cartesian axis. This complicates taking numerical derivatives. Consider for example a quantity u on the grid of Figure 4.1. We will consider the x -derivative of u in the point (x_1, z_1) . As we cannot determine this derivative directly, we will construct it from the derivatives along the grid axes. The vertical grid axis is called ξ , the other axis is called χ . We then have

$$u_\chi = u_x x_\chi + u_z z_\chi. \quad (4.1)$$

Re-ordering this yields expression we find

$$u_x = (u_\chi - u_z z_\chi) x_\chi^{-1}. \quad (4.2)$$

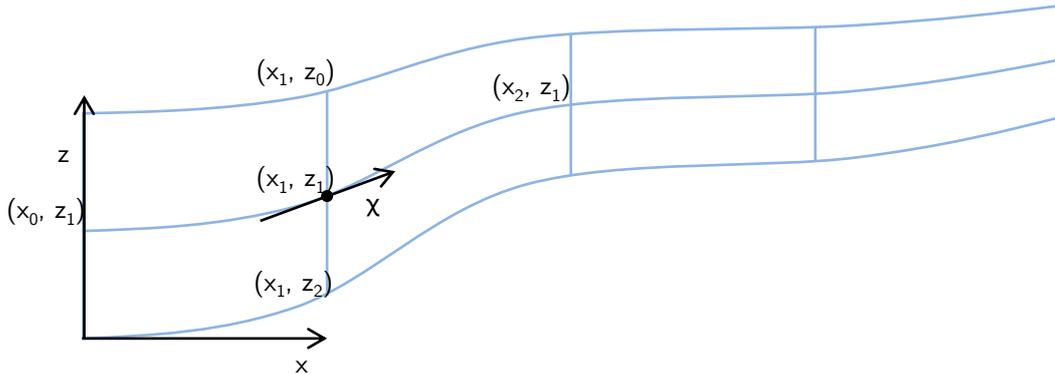


Figure 4.1: Grid with contraction of the z -axis in x -direction. The along-grid directions are (χ, z) , in the Cartesian coordinate system (x, z) .

Numerically, we calculate $u_x x_\chi^{-1}$ in one step by using a first order Taylor expansion for $x(\chi)$, so that $x_\chi = \frac{\Delta x}{\Delta \chi}$. This first-order expansion is reasonable if the grid slopes mildly with respect to the axes. For example using central derivatives this becomes

$$u_x(x_1, z_1) x_\chi^{-1}(x_1, z_1) = \frac{1}{2\Delta \chi} (u(x_2, z_1) - u(x_0, z_1)) \frac{\Delta \chi}{\Delta x} = \frac{1}{2\Delta x} (u(x_2, z_1) - u(x_0, z_1)). \quad (4.3)$$

Note that $\Delta\chi$ cancels from the equation. $z_\chi x_\chi^{-1}$ is computed along the same lines. For simplicity of notation we therefore define

$$u_{\bar{x}} = u_\chi x_\chi^{-1}$$

and write

$$u_x = u_{\bar{x}} - u_z z_{\bar{x}}. \quad (4.4)$$

The second derivative is found by expanding $u_{\chi\chi}$

$$\begin{aligned} u_{\chi\chi} &= u_{x\chi} x_\chi + u_x x_{\chi\chi} + u_{z\chi} z_\chi + u_z z_{\chi\chi} \\ &= (u_{xx} x_\chi + u_{xz} z_\chi) x_\chi + u_x x_{\chi\chi} + (u_{xz} x_\chi + u_{zz} z_\chi) z_\chi + u_z z_{\chi\chi}. \end{aligned}$$

Reordering yields

$$u_{xx} x_\chi x_\chi = u_{\chi\chi} - 2u_{xz} z_\chi x_\chi - u_x x_{\chi\chi} - u_{zz} z_\chi^2 - u_z z_{\chi\chi},$$

which is further rewritten to

$$u_{xx} = u_{\bar{x}\bar{x}} - 2u_{xz} z_{\bar{x}} - u_{zz} z_{\bar{x}}^2 - u_z z_{\bar{x}\bar{x}} - u_x x_{\chi\chi} x_\chi^{-2}.$$

We simplify this by assuming

$$x_{\chi\chi} x_\chi^{-2} = x_{\bar{x}\bar{x}} \approx 0.$$

We thus find

$$u_{xx} = u_{\bar{x}\bar{x}} - 2u_{xz} z_{\bar{x}} - u_{zz} z_{\bar{x}}^2 - u_z z_{\bar{x}\bar{x}}. \quad (4.5)$$

4.4.2 Numerical integration

NiFTy also includes a function for integration:

■ Code sample 4.6 — integration.

```

1 integration(u, dim, low, high, grid, (optional) indices, INTMETHOD=None)
2 '''
3 Takes the derivative of u along dimension dim
4 Parameters:  u: array
5              Array to take derivative of.
6              dim: string or int
7              String name of the dimension with respect to with the derivative
8              needs to be taken. The name should match the name definition
9              in the grid dimensions variable. Alternatively it can be the integer
10             position of this dimension is the grid dimensions list.
11             low: int
12             Grid index of dimension dim that is used as lower integration bound
13             high: int, list or 1D array
14             Grid index or list/array of indices of dimension dim that is used as upper
15             integration bound
16             grid: DataContainer instance
17             DataContainer containing a full grid variable (with all subvariables).
18             see explanation of derivative for more information
19             indices: (optional) arrays or numbers
20             One or multiple arguments stating the grid-indices along which the
21             output should be returned (as in DataContainer). If not set, returns
22             output along the entire grid
23             INTMETHOD: (optional) string
24             Numerical integration scheme, see below for options.
25             If None, the derivation method is taken from the
26             iFlow standard configuration file (keyword DERMETHOD).
27 '''

```

The returned array has the same shape as the original input, but the size of the dimension along which it is integrated has the same size as the argument 'high'.

The integrate function allows for three quadrature rules:

1. Trapezium rule (or first-order Newton-Cotes integration) (keyword TRAPEZIUM).
2. Simpson's rule (or second-order Newton-Cotes integration) (keyword SIMPSON).
3. A Simpson-like quadrature using interpolation (keyword INTERPOLSIMPSON).

A closed (i.e. including endpoints) Newton-Cotes quadrature can in general be described as

$$\int_{x_n}^{x_{n+N}} f dx \approx \frac{x_{n+N} - x_n}{N} \sum_{i=0}^N w_i f(x_i), \quad (4.6)$$

where N is the order of the quadrature and w_i are the weights, depending on N and on the uniformity of the grid. Below we will discuss the implementation of the three methods in NiFTy with regard to this quadrature formulation.

The trapezium rule has $N = 1$ and $w_0 = w_1 = 0.5$ regardless of the grid. It equals integrating a simple linear interpolation of the discrete function values. The quadrature (4.6) now describes a primitive function; i.e. it can also be written as

$$F(x_{n+1}) \approx \frac{x_{n+N} - x_n}{N} \sum_{i=0}^N w_i f(x_i)$$

where F is a primitive function of f for $n > 0$ and we choose $F(x_0) = 0$ (by virtue of choosing the constant of integration).

In the implementation, we compute the primitive in a separate (public) NiFTy function 'primitive'. This only uses addition and multiplication of numpy arrays and is very fast. The primitive is then summed from $F(x_0)$ to any (range of) other x values in the function 'integrate'. This process is most time consuming

The Simpson rule uses $N = 2$ and the weights depend on the grid spacing. The quadrature cannot be computed for the second grid point x_1 (which would require x_{-1}). Instead, the method uses an average of 1) the Simpson rule over all but the second grid point, where it uses the trapezoidal rule and 2) the Simpson rule in reverse integration direction over all but the second-to-last grid point, where it uses trapezoidal rule. The Simpson rule does not describe a simple primitive function of non-overlapping grid intervals. So, using the primitive function for integration requires careful bookkeeping of which elements to sum-up. An implementation using such a primitive has been tried, but is very slow. This is because the computation of the weights using the scipy function 'newtoncotes' is not vectorised and thus slow (i.e. requires for-loops).

The implementation therefore uses a scipy integration library 'sims' that computes the integral immediately. Downside is that the integration needs to be repeated for every upper bound, so that integration from x_0 to multiple x values scales linearly with the number of upper bounds.

As an alternative, we have implemented a Simpson-like scheme using an interpolating function. This method uses a refined grid \tilde{x} with one new grid point added in the middle of each grid cell. The function value in these intermediate points is determined using a quadratic interpolation and is called \tilde{f} . We can then apply the Simpson rule on this new

grid to calculate a primitive function in the original grid points. In an equation this reads

$$\begin{aligned} \int_{x_n}^{x_{n+1}} f dx &= \int_{\tilde{x}_{2n}}^{\tilde{x}_{2n+2}} \tilde{f} d\tilde{x} \approx \frac{\tilde{x}_{2n} - \tilde{x}_{2n+2}}{2} (w_0 \tilde{f}(\tilde{x}_{2n}) + w_1 \tilde{f}(\tilde{x}_{2n+1}) + w_2 \tilde{f}(\tilde{x}_{2n+2})) \\ &= \frac{x_n - x_{n+1}}{2} (w_0 f(x_n) + w_1 \tilde{f}(\tilde{x}_{2n+1}) + w_2 f(x_{n+1})). \end{aligned}$$

This method has two advantages. Firstly it describes a simple primitive function using non-overlapping intervals. Secondly, the weights are fixed because, by definition, \tilde{x}_{2n+1} is always exactly in the middle of the interval. The weights equal $w_0 = w_2 = 1/3$ and $w_1 = 4/3$.

We will consider the computation time and accuracy of the integral of the function $f = 1/(x+0.01)$ on the interval $x \in [0, 1]$ on a grid with 100 and 500 grid cells. The experiment is repeated 30 times to get accurate results for the computation time. We compute the integral $\int_0^{x_i} f dx$, where x_i indicates the grid points. We consider integrating up to all indices separately and integrating only up to $x_{j_{\max}} = 1$. The results are presented in Table 4.1. Note that the error is measured for the case integrating up to all separate indices and we take the sup-norm.

	100 cells			500 cells		
	Error	Time (msec) $i \in [0, j_{\max}]$	Time (msec) $i = j_{\max}$	Error	Time (msec) $i \in [0, j_{\max}]$	Time (msec) $i = j_{\max}$
Trapezoidal	0.077	1	1	$3.3 \cdot 10^{-3}$	5	4
Simpson	0.013	61	2	$3.2 \cdot 10^{-4}$	718	6
Interpolated Simpson	0.013	13	13	$4.9 \cdot 10^{-5}$	320	320

Table 4.1: Computation time and accuracy of numerically integrating the function $f = 1/(x+0.01)$ using three numerical integration routines in `NUMPY`. The computation time is summed over 30 repetitions.

The Simpson rule is about one order of magnitude more accurate than the trapezoidal rule, but also takes up to two orders of magnitude more computational time. The interpolated Simpson rule is as accurate as the Simpson rule and even much more accurate on high resolutions. Also, it is significantly faster when computing the integral with all grid points as upper boundaries. However, it is relatively slow for computing the integral for only one upper boundary point. This is because the interpolation of the function takes a fixed amount of time.

4.4.3 Grids and axes

In order to facilitate the construction of numerical grids, `iFlow` provides the function `Make a grid`
`makeRegularGrid`

■ Code sample 4.7 — `makeRegularGrid`.

```

1 makeRegularGrid(dimensions, axisType, axisSize, axisOther, enclosures, contraction = None, copy = None)
2 '''Make separate regular grid axes and enclosures for all dimension.
3
4 Parameters:   dimensions: str or list of str
5               dimension names
6               axisType: str or list of str
7               Supports several types, see Section on grids
8               axisSize: int or list of int
9               length of axis
10              axisOther:
11                  other arguments, depending on the axis type (see Section on grids)
12              enclosures: tuple or list of tuples
13                  One tuple for each axis. A tuple consists of two elements:

```

```

14         lower and upper boundary (scalar, array or function), where
15         lower corresponds to the dimensionless 0 point and upper to
16         the dimensionless 1 point
17     contraction: list of length len(dimension)
18         of lists of length of len(dimension), optional. Requires one sublist
19         per dimension. Sublist i contains the dimension names on which
20         the enclosure of dimension i depends. Assumes empty lists if not
21         provided
22     copy: list of length len(dimension)
23         list containing elements 0 and/or 1. Element 1 indicates that lower-
24         dimensional arrays will automatically be copied over the dimension
25         belonging to the position of the element. A zero indicates the lower-
26         dimensional array will only be copied to the first element of this
27         dimension, appended by zeros. By default, copying is turned on
28         (vector of ones). A zero is useful for e.g. a dimension with harmonic
29         components, where a value should only be copied to the zero-
30         frequency component.
31     '''

```

dimensions of axes The grid axes are saved in a dimensionless form with values between 0 and 1. The dimensional axes can be retrieved by combining the dimensionless axes and the grid enclosures. This is simplified by the NiFTy function `dimensionalAxis`

■ Code sample 4.8

```

1 dimensionalAxis(grid, dim, args, kwargs)
2 '''Returns the dimensional form of the axes of dimension dim
3
4 Parameters:   grid: DataContainer
5               DataContainer instance containing a full iFlow grid
6               dim: int or str
7               Name or number of the dimension of the axis of interest
8               args: tuple of integers or arrays
9                 Grid indices to return grid on
10              kwargs: dict of pairs dimname: int or array
11                    Dimensionless coordinates to return grid on
12 '''

```

4.4.4 Timing

NiFTy provides the `Timer` class to track computation times. The `Timer` class has the methods

- `tic()` start the timer
- `toc()` stop the timer and add elapsed time to the total time registered in this timer instance
- `reset()` reset this timer instance to zero
- `disp(message)` print elapsed time preceded by the message and a semicolon
- `string(message)` as `disp`, but then returns a string instead of printing it

As these elapsed time between each `tic` and `toc` of a `Timer` instance is saved in a class variable, multiple timers may be used within the same script to time different components. Also a single `Timer` instance may be passed from one function or class to another.

4.4.5 Operations on harmonic components

As many of the standard modules in iFlow work with quantities in harmonic components, NiFTy implements a number of tools to deal with common operations on harmonic components. Consider functions $u(t)$ and $v(t)$ of time t . We will write these functions in terms of a finite set of $p + 1$ harmonic components

$$u(t) = \text{Re} \left(\sum_{n=0}^p \hat{u}_n e^{i\omega t} \right),$$

$$v(t) = \text{Re} \left(\sum_{n=0}^p \hat{v}_n e^{i\omega t} \right),$$

for some angular frequency ω and complex coefficients \hat{u}_n and \hat{v}_n . The complex coefficients can be written as numerical vectors \hat{u} and \hat{v} with elements \hat{u}_n and \hat{v}_n .

The product of two time series $w(t) = u(t)v(t)$ again yields a time series $w(t)$ that can be written as a vector of its harmonic components \hat{w} . This may be computed as

$$\hat{w} = \text{complexAmplitudeProduct}(\hat{u}, \hat{v}, 0).$$

The NiFTy function `complexAmplitudeProduct` takes two vectors of harmonic components and the dimension number along which the harmonic components are given (here 0, because the vectors are 1D). Note that two vectors \hat{u} and \hat{v} with $p + 1$ components would yield a signal \hat{w} with potentially $2p + 1$ components. This is truncated to $p + 1$ components.

Similarly, $w(t) = u(t)v(t)$ may be written as the matrix equation $\hat{w} = \mathcal{U}\hat{v}$. The matrix \mathcal{U} is created using the NiFTy command `toMatrix`. We can thus write

$$\hat{w} = \text{arraydot}(\text{toMatrix}(\hat{u}), \hat{v}),$$

where `arraydot` is another function that extends the numpy dot product function to functions of more than two variables, where only the last dimensions should be considered in the matrix or matrix-vector product.

4.5 The sTeP toolbox

The sTeP (Standard Tools for Plotting) toolbox is created to quickly make standardised plots of all the available output data created by iFlow. The fact that, with simple single-line commands, the user can plot almost any combination of variables stored in the `dataContainer` or in an output file created by iFlow, highly simplifies the analysis of the results. Currently, the toolbox contains three types of plots, i.e. `lineplot`, `contourplot` and `transportplot_mechanisms`. The following sections present an overview of how to import and use the sTeP toolbox and presents a description of the three standardised plotting commands.

Plotting modules can be used in two ways. First, they may be included in an input file together with other modules that do computations. This way, the results of a computation can be plotted directly. Second, plot modules may be used in a separate input file and plot data that was saved before. In the latter case, the plot module needs to be accompanied by a module that loads saved data (e.g. `ReadSingle` or `ReadMultiple`, see manual on the general module package). In either way, the plot module may look the same, since it does not care whether its data comes from a file or directly from computations.

Plot modules

4.5.1 General

The sTeP toolbox provides two commands to standardise the way plots look. A minimum example is given below

■ Code sample 4.9

```

1  import step as st
2  from step import Step
3  import matplotlib.pyplot as plt
4
5  class Plot:
6      # Variables
7      logger = logging.getLogger(__name__)
8
9      # Methods
10     def __init__(self, input):
11         self.input = input
12         return
13
14     def run(self):
15         self.logger.info('Module Plot is running')
16
17         # configure plot settings
18         st.configure()
19
20         # some plot code, either using matplotlib ...
21         plt.figure(1, figsize=(1,2))
22         plt.plot( .... )
23
24         # ... or using STeP functions
25         step = Step.Step(self.input)
26         step.lineplot(...)
27
28         # display plots
29         st.show()

```

configure and
show

The sTeP toolbox is imported as `import step as st`, from which the functions `st.configure()` and `st.show()` can be called. The former function configures font, resolution and axes conform iFlow standards. The latter function replaces the `plt.show()` function of `matplotlib.pyplot` conform iFlow standards and corrects among others figure size, background color and tight layout.

figsize

sTeP follows a customised figure size scheme. For users of `matplotlib.pyplot` (here imported as `plt`) one can use this as

```
plt.figure(fignum, figsize=(v_units, h_units),
```

replacing `fignum` (figure number), `v_units` (vertical units) and `h_units` (horizontal units) by integers. Here, one vertical unit represents one standard height for one subplot. Two vertical units are used for figures with two subplots in the vertical direction etcetera. One or two horizontal units may be used. One horizontal unit represents a figure of 7 cm, which we define as half an A4 page or one column in a two-column layout. Two horizontal units then equals 14 cm or a full A4 width.

To use the standardised plotting functions, from the sTeP toolbox the `step` module needs to be imported by `from step import Step` and subsequently an object is instantiated by `step = Step.Step(self.input)`. Here, the argument `self.input` passes the data from the `dataContainer` to the `step` object, from which the data can be plotted by using any of the three plot functions described in the next sections.

4.5.2 Lineplot

To make a line plot of a certain variable over a certain axis, the `STeP` module provides the function `lineplot`

■ Code sample 4.10

```

1 lineplot(axis1, axis2, args, kwargs)
2 """ Makes a lineplot of a variable over a certain axes
3
4 Parameters:   axis1: string
5               name of the dimension (of grid) or variable to plot on the horizontal axis
6               axis2: string
7               name of the dimension (of grid) or variable to plot on the vertical axis
8               NB. of axis1 and axis2 one should be a grid dimension and one should be a variable
9               args: Optional, strings
10              optional names of subkeys of the variable in axis1 or axis2
11              grid dimensions: coordinates (between 0 and 1 for non-integer axes) in a single
12              number or array
13              grid dimensions other than in axis1 or axis2. For a grid
14              with axes 'x', 'z', 'f' with a plot over dimension 'x', one can specify e.g.
15              z=[0, 1] and f=0. This will generate 2 lines, for f=0, z=0 and f=0, z=1
16              which are plotted in one figure or subplots (see argument subplot below)
17              sublevel: boolean
18              show sub-level data or not, i.e. separate contributions under subkeys of the
19              variable to plot. If True, the sub-level data is either shown in one figure or
20              in subplots as specified by the argument subplot (see below)
21              subplots: string
22              Allows for 'sublevel' or the string name of a grid dimension. If there
23              are more sublevels or more lines for the grid dimensions (see above),
24              the argument subplots places this in subplots rather than multiple lines
25              in one figure.
26              plotno: integer
27              plot number
28              operation: python function
29              makes an operation on the physical variable using the python function.
30              This function could be for instance a numpy function (e.g. np.abs,
31              np.angle, np.imag or np.real) or a nifty function (e.g. ny.scalemax)
32 """

```

4.5.3 Contourplot

To make a 2DV contour plot of a certain variable over two axes, the `STeP` module provides the function `contourplot`

■ Code sample 4.11

```

1 contourplot(axis1, axis2, value_label, args, kwargs)
2 """ Plots 2DV contourplots of a variable over two axes
3
4 Parameters:axis1: string
5             name of the grid dimension to plot on the horizontal axis
6             axis2: string
7             name of the grid dimension to plot on the vertical axis
8             value_label: string
9             name of the physical variable for which the controurplot needs to be made
10            args: Optional, strings
11            optional names of subkeys of the variable in value_label
12            grid dimensions: coordinates (between 0 and 1 for non-integer axes) in a single
13            number or array
14            grid dimensions other than in axis1 and axis2. For a grid
15            with axes 'x', 'z', 'f' with a plot over dimension 'x', one can specify e.g.
16            z=[0, 1] and f=0. This will generate 2 lines, for f=0, z=0 and f=0, z=1
17            which are plotted in one figure or subplots (see argument subplot below)
18            sublevel: boolean
19            show sub-level data or not, i.e. separate contributions under subkeys of the
20            variable to plot. If True, the sub-level data is either shown in one figure or
21            in subplots as specified by the argument subplot (see below)
22            subplots: string
23            Allows for 'sublevel' or the string name of a grid dimension. If there
24            are more sublevels or more lines for the grid dimensions (see above),
25            the argument subplots places this in subplots rather than multiple lines
26            plotno: integer
27            plot number
28            operation: python function
29            makes an operation on the physical variable using the python function.
30            This function could be for instance a numpy function (e.g. np.abs,
31            np.angle, np.imag or np.real) or a nifty function (e.g. ny.scalemax)
32 """

```

■

4.5.4 Transportplot mechanisms

To plot the transport mechanisms calculated by the sediment modules of the `numerical2DV` and `semi_analytical2DV` packages, the `sTep` module provides the function `transport_mechanisms`

■ Code sample 4.12 — transportplot mechanisms.

```

1 transportplot_mechanisms(kwargs)
2 """ Plots the advective transport based on the physical mechanisms that force it.
3
4 Parameters:  sublevel: string
5             displays underlying levels of the associated mechanisms: 'sublevel',
6             'subsublevel' or False
7             plotno: integer
8             plot number
9             display: integer or list of strings
10            displays the underlying mechanisms indicated. An integer
11            plots the largest contributions up to that integer and a list
12            of strings plots the mechanisms in that list
13            scale: boolean
14            scales the transport contributions to the maximum value
15            of all contributions
16            concentration: boolean
17            plots the depth-mean, sub-tidal concentration in the background

```

18 """

■



A. Object-oriented programming

This chapter is a short introduction to object-oriented programming. Focus is on the basic concept and terminology necessary for understanding iFlow. The explanation here is largely independent of the programming language, but we will give a few hints to the names used in Python.

In object-oriented programming, a program consists of interacting objects. Objects are instantiated from *classes*. A class is the programming script that contains all functions or instructions a specific object might have. While running a program, objects are instantiated, meaning that we make an actual object that inherits all the functions or instructions of the class. For example one can define a class `Calculator` (class names are capitalised by convention) with a set of functionalities. During a simulation we may make two instances of a `Calculator`, e.g. `myCalculator` and `yourCalculator`. These instances work independently. Therefore if we make a computation using `myCalculator`, then the `yourCalculator` does not have any notion of this.

Classes consist of functionalities, called *methods*, and knowledge, called *class variables*. Methods are like functions. They can take arguments and return results. Common practice is that methods are either *queries* or *assignments*. Queries do not set class variables and preferably do little computations, however they return some variable. Assignments on the other hand can set class variables and do extensive computations, but do not return variables. The reasons for this are explained below. Class variables are the global knowledge of a class, so that a class variable set in one method of a class can be used in another method of the same class.

A class is obliged to contain a constructor (`__init__` in Python). This is a method that is invoked when an object is instantiated from a class and may contain any set of instructions to set up the object. An instance is created by implicitly calling the constructor. In Python this is done as

```
myCalculator = Calculator().
```

This automatically calls the `__init__` method of `Calculator` and creates a new instance.

Different objects interact by calling each other's methods. It is common practice to do this using a strict hierarchy. That is, if Object A calls a method of Object B, Object B should be seen as a subordinate of Object A and should not call any method of Object A (i.e. a subordinate does not tell his boss what to do). In many cases this is very intuitive. For example consider two classes `Student` and `Calculator`. Clearly it should be the student that calls methods of `Calculator` and it would be odd if `Calculator` were to call any methods of `Student`. Along the same lines one should prevent triangle-relationships, i.e. If Object A calls a method of Object B which in turn calls a method of Object C, Object C should not call a method of Object A (i.e. else it would be as if a junior employee tells the CEO what to do). In designing a code, it may sometimes be tempting to violate the rules of the strict hierarchy. However, this ultimately leads to a code that becomes unwieldy and difficult to track. In almost all cases, strictly hierarchical designs are possible.

Another important concept in object-oriented programming is *responsibility*. Responsibility relates to the idea that an object should have a clear and preferably intuitive set of tasks and knowledge. In the example of the calculator it makes sense that this class has functions to make computations, but it would not make sense if it had a method `sumTwoNumbers`, where it makes up two numbers itself and sums them. Finding the numbers to put into a calculator is not the responsibility of the calculator itself. Likewise it would not make sense if the calculator had a class variable `weather`, which stores the current state of the weather. Thinking about responsibility of an object early in the design of a code helps to make intuitively understandable and meaningful classes. As a rule-of-thumb the design of code needs to be revisited if it is difficult to find a short name for a class, if a class name does not cover all tasks/knowledge of the class or if the class name becomes too conceptual or is not a noun. Such problems typically point to a problem with class responsibility.

From the concept of responsibility it follows that objects are responsible for what they know, i.e. their class variables. Therefore an object should not set or change the class variable of another object. Common practice is even that an object should not directly see the class variables of another object. Instead, a class variable may be seen through invoking a method. In the example of the calculator, we could add a method `getResult()`, that returns the result. This way an object is responsible for what portion of its knowledge it wants to disclose. To see how this helps program design, consider two classes `Manager` and `Researcher`. Say that `Researcher` has a method `doResearch()` that results in the class variables `allDetails` and `mainResults`. Intuitively it makes sense that `Researcher` has a method `returnMainResults`, but does not disclose the variable `allDetails`. This is helpful, as it is not the responsibility of `Manager` to know all the details of a research project. Therefore not offering the temptation by not disclosing `allDetails` helps to obey the responsibilities. This is one of the reasons why methods are often strictly separated into queries and assignments (see above). Assignments generate knowledge. This knowledge is owned by the object that generated it and should not automatically be given to another object¹.

Related to the concept of responsibility is the concept of contracts. Contracts are typically focussed on method level and concern arrangements between the caller of a method and the owner of a method. Again consider the example of the student and the calculator. The implicit contract between them is that `calculator` promises to function normally if the input from `student` makes sense. `calculator` cannot promise anything if the input is nonsense. This means that, in case of nonsense input, the calculator may either seemingly work fine or crash. As a consequence of this contract `calculator` does not have to check if the input to its methods makes sense. Similarly, `student` does not have to check if the output of `calculator` is correct. He can trust it is correct if his input makes sense. This potentially saves a lot of programming work spent on checking. The terms of the contract of each method are written in comment lines directly under the method definition, called the *docstring*.

¹Note that the strict separation between queries and assignments is purposefully ignored in many places in iFlow, most notably in the `run` method of a module

The docstring provides a short description of the method, the input data and data types expected and the output data and data types that may be expected if the input is correct. To clearly see the relation between contracts and responsibilities it is helpful to remember that a subordinate object has the responsibility to do his work well assuming his superior (i.e. the object that calls his method) keeps his responsibility and gives a correct command.

There are a view cases where reasonable input may lead to a crash of a method. An example is when `Student` uses `Calculator` to compute $0/0$. While the calculator could demand from the student that the denominator is non-zero, it can be argued that this is too much to ask of `Student`. In this case it is allowed that `Calculator` does not pose the non-zero input requirement and crashes when the denominator of a division is zero. The `Calculator` should however foresee this and should provide a clear error message upon crashing.

Many more things might be said about object-oriented programming, but this gives a sufficient background for understanding this manual and understanding `iFlow`. For interested readers, next topics to look up include subclasses and public versus private.



Bibliography

Brouwer, R. L. (2017). *Semi-analytical 2DV perturbation model. Package for iFlow.*

Dijkstra, Y. M. (2017). *iFlow modelling framework. User manual & technical description.*

Dijkstra, Y. M., Brouwer, R. L., Schuttelaars, H. M., and Schramkowski, G. P. (Manuscript submitted to Geoscientific Model Development). The iFlow Modelling Framework v2.4. A modular idealised process-based model for flow and transport in estuaries.